

# 8 bits de PODER

On your AMSTRAD CPC



"Limitations are not a problem, but a source of inspiration".

**V42\_00**

Jose Javier García Aranda



## INDEX

<b>1 WHY PROGRAM A 1984 MACHINE TODAY? .....</b>	<b>9</b>
<b>2 8BP FUNCTIONS AND MEMORY USAGE .....</b>	<b>11</b>
2.1 WHAT IS AN RSX LIBRARY? .....	12
2.2 FUNCTIONS OF 8BP .....	13
2.3 AMSTRAD CPC ARCHITECTURE .....	14
2.3.1 <i>GOSUB /RETURN Stack</i> .....	19
2.3.2 <i>An experiment to see the ROM with Winape</i> .....	20
2.4 8BP MEMORY MAP AND ASSEMBLY OPTIONS.....	21
<b>3 TOOLS NEEDED.....</b>	<b>25</b>
<b>4 FIRST STEPS WITH 8BP .....</b>	<b>27</b>
4.1 INSTALL WINAPE .....	27
4.2 GETTING FAMILIAR WITH WINAPE: "HELLO WORLD" .....	27
4.3 DOWNLOAD THE 8BP LIBRARY .....	27
4.4 RUN THE DEMOS.....	28
4.5 CREATING YOUR FIRST PROGRAMME WITH 8BP .....	29
4.6 CREATE YOUR .DSK WITH YOUR 8BP GAME .....	32
<b>5 STEPS YOU NEED TO TAKE TO MAKE A GAME .....</b>	<b>33</b>
5.1 DIRECTORY STRUCTURE OF YOUR PROJECT .....	33
5.2 YOUR GAME IN JUST 3 FILES.....	34
5.3 CREATE A DISC OR TAPE WITH YOUR GAME .....	36
5.3.1 <i>Making a disc</i> .....	36
5.3.2 <i>Making a ribbon with Winape</i> .....	36
5.3.3 <i>Make a tape easily with CPCDiskXP, 2cdt and tape2wav</i> .....	38
5.3.4 <i>Troubleshooting LOAD and MEMORY</i> .....	39
<b>6 LIBRARY ASSEMBLY, MUSIC AND GRAPHICS.....</b>	<b>42</b>
6.1 MAKE_ALL.ASM.....	43
6.2 STRUCTURE OF THE IMAGE FILE .....	45
6.3 ANIMATION SEQUENCE FILE STRUCTURE.....	45
6.4 STRUCTURE OF THE ROUTING FILE.....	46
6.5 STRUCTURE OF THE WORLD MAP FILE .....	46
<b>7 CYCLE OF PLAY .....</b>	<b>47</b>
7.1 HOW TO MEASURE THE FPS OF YOUR GAME CYCLE .....	47
<b>8 SPRITES.....</b>	<b>49</b>
8.1 EDITING SPRITES WITH SPEDIT AND ASSEMBLING SPRITES .....	49
8.2 PRINT A SPRITE .....	54
8.3 SPRITE FLIPPING .....	55
8.4 SPRITES WITH OVERWRITING .....	57
8.4.1 <i>Using overwrite to improve sprite overlaps</i> .....	61
8.4.2 <i>Overwriting with 4 background colours</i> .....	62
8.4.3 <i>Overwrite in MODE 1</i> .....	63
8.4.4 <i>How to paint sprites "from behind" the background</i> .....	64
8.4.5 <i>How to use more colours with overwriting</i> .....	65
8.5 SPRITES WITH BACKGROUND IMAGES .....	67

8.6	TABLE OF SPRITE ATTRIBUTES .....	69
8.7	PRINTOUT OF ALL SPRITES AND SORTING .....	74
8.8	COLLISIONS BETWEEN SPRITES .....	76
8.9	SPRITE COLLISION SENSITIVITY ADJUSTMENT .....	78
8.10	WHO COLLIDES AND WITH WHOM: COLSPALL .....	79
8.10.1	<i>How to programme a multi-shot using COLSPALL</i> .....	80
8.10.2	<i>Who collides when there are several overlaps</i> .....	81
8.10.3	<i>Advanced use of status byte in collisions</i> .....	83
8.11	TABLE OF ANIMATION SEQUENCES .....	84
8.12	SPECIAL ANIMATION SEQUENCES .....	85
8.12.1	<i>Death sequences</i> .....	86
8.12.2	<i>End sequences</i> .....	87
8.12.3	<i>Chained sequences</i> .....	87
8.12.4	<i>Animation macro-sequences</i> .....	87
<b>9</b>	<b>YOUR FIRST SIMPLE GAME .....</b>	<b>91</b>
9.1	NOW, LET'S JUMP! BOING, BOING! .....	91
<b>10</b>	<b>SCREEN SETS: LAYOUT OR TILE MAPS .....</b>	<b>95</b>
10.1	LAYOUT DEFINITION AND USE.....	95
10.2	EXAMPLE OF A GAME WITH LAYOUT .....	98
10.3	HOW TO OPEN A GATE IN THE LAYOUT .....	100
10.4	A PUZZLE GAME: LAYOUT WITH A BACKGROUND .....	101
10.5	HOW TO SAVE MEMORY IN YOUR LAYOUTS .....	103
<b>11</b>	<b>ADVANCED PROGRAMMING AND "BULK LOGICS" .....</b>	<b>105</b>
11.1	MEASUREMENT OF THE SPEED OF COMMANDS .....	105
11.2	MAKE A SINGLE LOGIC TO RULE ALL YOUR DISPLAYS .....	112
11.3	MASS LOGIC" TECHNIQUE.....	113
11.3.1	<i>Moves 32 sprites with massive logics</i> .....	114
11.3.2	<i>Alternating and periodic cascade execution</i> .....	115
11.3.3	<i>Simple example of mass logic</i> .....	117
11.3.4	<i>Block" movement of squadrons</i> .....	118
11.3.5	<i>Massive logic technique in "pacman" type games</i> .....	119
11.3.6	<i>Reduction of number of instructions in set cycle</i> .....	121
11.3.7	<i>Routes that accelerate the game by manipulating the state</i> .....	125
11.3.8	<i>Routing sprites with "massive logics"</i> .....	125
<b>12</b>	<b>COMPLEX PATHS: ROUTEALL COMMAND.....</b>	<b>129</b>
12.1	PLACES A SPRITE IN THE MIDDLE OF A ROUTE : ROUTESP.....	131
12.2	CREATING ADVANCED ROUTES .....	133
12.2.1	<i>Forced state changes from routes</i> .....	133
12.2.2	<i>Forced sequence changes from routes</i> .....	135
12.2.3	<i>Forced image changes from routes</i> .....	135
12.2.4	<i>Forced rerouting from routes</i> .....	139
12.2.5	<i>Forced path changes from BASIC</i> .....	139
12.2.6	<i>Forced animation from routes</i> .....	141
12.2.7	<i>How to build "dynamic" (not predefined) routes</i> .....	141
12.2.8	<i>Programming of routes including patterns</i> .....	142
12.2.9	<i>Typology of routes</i> .....	143

<b>13</b>	<b>HALF-BYTE SMOOTH MOTION.....</b>	<b>145</b>
<b>14</b>	<b>SCROLLING GAMES .....</b>	<b>147</b>
14.1	STARS: SCROLL OF STARS OR SPECKLED EARTH .....	147
14.2	SCROLL USING MOVERALL AND/OR AUTOALL .....	150
14.3	TECHNIQUE OF "SPOTTING" .....	152
14.4	MAP2SP: SCROLL BASED ON A WORLD MAP .....	155
14.4.1	<i>Map of the World (Map Table)</i> .....	157
14.4.2	<i>Using the MAP2SP function.</i> .....	158
14.4.3	<i>Example of a phase file</i> .....	161
14.4.4	<i>Enemy collision with map</i> .....	163
14.4.5	<i>Background images in your scroll</i> .....	164
14.5	PARALLAX SCROLL .....	165
14.6	DYNAMIC MAP UPDATE:  UMAP .....	166
14.7	ANIMATION AND INK SCROLLING: RINK COMMAND .....	168
14.7.1	<i>2D car racing</i> .....	169
14.7.2	<i>Brick Scroll</i> .....	171
<b>15</b>	<b>PLATFORM GAMES .....</b>	<b>173</b>
<b>16</b>	<b>HORDES OF ENEMIES IN SCROLLING GAMES.....</b>	<b>175</b>
<b>17</b>	<b>REDEFINABLE MINI-CHARACTERS: PRINTAT .....</b>	<b>177</b>
17.1	CREATE YOUR OWN MINI-CHARACTER ALPHABET .....	178
17.2	DEFAULT ALPHABET FOR MODE 1 .....	179
<b>18</b>	<b>PSEUDO 3D .....</b>	<b>181</b>
18.1	3D PROJECTION .....	183
18.1.1	<i>Mathematics of pseudo 3D projection</i> .....	184
18.1.2	<i>Curves</i> .....	188
18.2	ZOOM IMAGES .....	189
18.3	USE OF SEGMENTS.....	191
<b>19</b>	<b>MUSIC .....</b>	<b>193</b>
19.1	EDITING MUSIC WITH WYZ TRACKER .....	193
19.2	ASSEMBLING THE SONGS.....	194
19.3	WHAT TO DO IF YOU CAN'T FIT MUSIC IN 1400 BYTES.....	195
<b>20</b>	<b>C PROGRAMMING WITH 8BP .....</b>	<b>197</b>
20.1	FIRST STEP: PROGRAM YOUR BASIC GAME .....	198
20.2	STEP 2: TRANSLATE YOUR BASIC GAME CYCLE INTO C .....	199
20.2.1	<i>GOSUB and RETURN in C</i> .....	203
20.2.2	<i>BASIC to C communication with BASIC variables</i> .....	204
20.2.3	<i>BASIC and C text strings</i> .....	207
20.3	THIRD STEP: COMPILE USING "COMPILA.BAT".....	208
20.4	STEP 4: CHECK MEMORY LIMITS .....	210
20.5	FIFTH STEP: LOCATE THE ADDRESS OF THE FUNCTION TO BE INVOKED FROM BASIC211	
20.6	STEP 6: INCLUDE IN YOUR .DSK GAME THE NEW BINARY .....	212
20.7	8BP FUNCTION REFERENCE IN C .....	213
20.8	BASIC FUNCTION REFERENCE IN C ("MINIBASIC") .....	215
<b>21</b>	<b>8BP LIBRARY REFERENCE GUIDE .....</b>	<b>217</b>

21.1	LIBRARY FUNCTIONS.....	217
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
22	HOW TO MAKE A SCOREBOARD .....	243
23	POSSIBLE FUTURE IMPROVEMENTS TO THE LIBRARY .....	245
23.1	MEMORY FOR LOCATING NEW FUNCTIONS .....	245
23.2	PIXEL RESOLUTION PRINTING .....	245
23.3	LAYOUT DE MODE 1 .....	245
23.4	FILMATION CAPACITY .....	245
23.5	HARDWARE SCROLL FUNCTIONS .....	246
23.6	MIGRATING 8BP LIBRARY TO OTHER MICROCOMPUTERS .....	247
24	SOME GAMES MADE WITH 8BP .....	249
24.1	MUTANT MONTOYA.....	249
24.2	ANUNNAKI, OUR ALIEN PAST .....	250
24.3	NIBIRU .....	251
24.4	FRESH FRUITS & VEGETABLES .....	252
24.5	"3D RACING ONE.....	252
24.6	SPACE PHANTOM .....	253
24.7	ETERNAL FROGGER.....	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, THE MAN .....	258

24.13	MINI GAMES .....	259
24.13.1	<i>Mini-pong</i> .....	259
24.13.2	<i>Mini-Invaders</i> .....	260
<b>25</b>	<b>APPENDIX I: VIDEO MEMORY ORGANISATION</b> .....	<b>261</b>
25.1	THE HUMAN EYE AND THE RESOLUTION OF THE CPC .....	261
25.2	VIDEO MEMORY .....	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>Display memory</i> .....	262
25.3	CALCULATION OF A SCREEN ADDRESS .....	264
25.4	SCREEN SWEEPS .....	264
25.5	HOW TO MAKE A LOADING SCREEN FOR YOUR GAME .....	265
<b>26</b>	<b>APPENDIX II: THE PALETTE</b> .....	<b>269</b>
<b>27</b>	<b>APPENDIX III: INKEY CODES</b> .....	<b>271</b>
<b>29</b>	<b>APPENDIX IV: AMSTRAD CPC ASCII TABLE</b> .....	<b>273</b>
<b>30</b>	<b>APPENDIX V: SOME SOUND EFFECTS</b> .....	<b>275</b>
<b>31</b>	<b>APPENDIX VI: INTERESTING FIRMWARE ROUTINES</b> .....	<b>277</b>
<b>32</b>	<b>APPENDIX VII: TABLE OF SPRITE ATTRIBUTES</b> .....	<b>279</b>
<b>33</b>	<b>APPENDIX VIII: 8BP MEMORY MAP</b> .....	<b>281</b>
<b>34</b>	<b>APPENDIX IX: AVAILABLE COMMANDS 8BP</b> .....	<b>283</b>
<b>35</b>	<b>APPENDIX X: 8BP ASSEMBLY OPTIONS</b> .....	<b>285</b>
<b>36</b>	<b>APPENDIX XI: RSX/CALL MAPPINGS</b> .....	<b>287</b>
<b>37</b>	<b>APPENDIX XII: 8BP FUNCTIONS IN C</b> .....	<b>289</b>
<b>38</b>	<b>APPENDIX XIII: MINIBASIC IN C</b> .....	<b>291</b>



# 1 Why program a 1984 machine today?

Because limitations are not a problem but a source of inspiration.

Limitations, whether of a machine or a human being, or in general of any available resource, stimulate our imagination to overcome them. The AMSTRAD, a 1984 machine based on the Z80 microprocessor, has a small memory (64KB) and a small processing capacity, but only compared to today's computers. This machine is actually a million times faster than the one Alan Turing built to decipher the messages of the enigma machine in 1944. Like all computers of the 1980s, the AMSTRAD CPC booted up in less than a second, with the BASIC interpreter ready to receive user commands, BASIC being the language with which programmers learned and made their first developments. The AMSTRAD BASIC was particularly fast compared to its competitors, and aesthetically it was a very attractive computer!



*Fig. 1. The legendary AMSTRAD model CPC464*

As for the Z80 microprocessor, it is not even capable of multiplying (in BASIC you can multiply, but that is based on an internal program that implements multiplication by means of additions or register shifts), it can only do additions, subtractions and logical operations. Despite this, it was the best 8-bit CPU and only consisted of 8500 transistors, unlike other processors such as the M68000 whose name comes precisely from having 68000 transistors.

CPU	Number of transistors	MIPS (million instructions per second)	Computers and consoles that incorporate it
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 per core)	Raspberry pi 2, Nintendo 3DS, Samsung galaxy, ...
Intel i7 (5th generation)	2.600.000.000	238310 @ 3Ghz (almost! 500.000 times faster than a Z80 )	PC
AMD Ryzen 9 3950X (16 core)	3.800.000.000	749070 @4.8Ghz (1.3 million times faster than Z80)	PC

**Table 1Comparison of MIPS**

This makes programming extremely interesting and stimulating in order to achieve satisfactory results. All our programming must be geared towards reducing spatial (memory) and temporal (operations) computational complexity, forcing us to invent tricks, ruses, algorithms, etc., and making programming an exciting adventure. This is why the programming of machines with low processing capacity is a timeless concept, not subject to fashions or conditioned by the evolution of technology.

All the code for this book, including the library for you to make your own games or make contributions to the library, can be found in the GitHub project "8BP", at this URL. Just download the zip (in a later chapter I'll give you a step-by-step)

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

There is also a blog with a lot of information at:

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

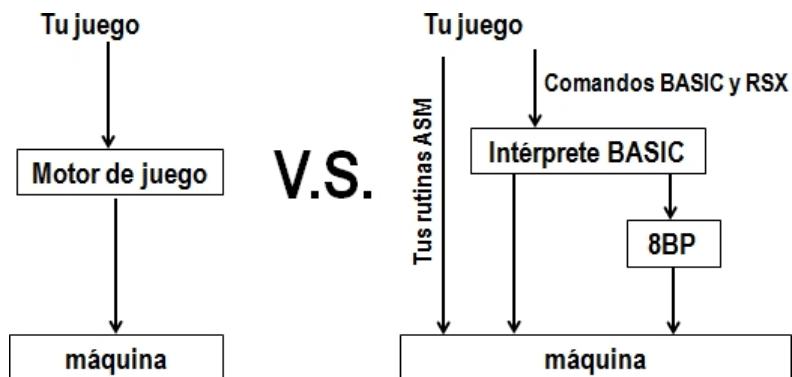
And a youtube channel:

[https://www.youtube.com/channel/UCThvesOT-jLU\\_s8a5ZOjBFA](https://www.youtube.com/channel/UCThvesOT-jLU_s8a5ZOjBFA)

## 2 8BP functions and memory usage

The 8BP library is not a "game engine". It is something in between a simple BASIC command extension and a game engine.

Game engines such as game-maker, AGD (Arcade Game Designer), Unity, and many others, limit to some extent the imagination of the programmer, forcing him to use certain structures, to program in a limited scripting language the logic of an enemy, to define and link game screens, etc.



*Fig. 2 Play Engines versus 8BP*

The 8BP library is different. It is a library capable of executing quickly what BASIC can't do. Things like printing sprites at full speed, or moving banks of stars around the screen, are things that BASIC can't do, and 8BP does it.

And it takes up only 8 KB!

BASIC is an interpreted language. This means that every time the computer executes a program line, it must first verify that it is a valid command by comparing the command string with all valid command strings. It must then syntactically validate the expression, the command parameters and even the allowed ranges for the values of those parameters. In addition, it reads the parameters in text format (ASCII) and must convert them to numerical data. Once all this work is done, it proceeds with the execution. Well, all this work that is carried out in each instruction is what differentiates a compiled program from an interpreted program such as those written in BASIC.

By equipping BASIC with the commands provided by 8BP, it is possible to make professional quality games, since the game logic you program can be executed in BASIC while CPU intensive operations such as printing on screen or detecting collisions between sprites, etc. are carried out in machine code by the library. However, it's not all easy and trouble-free. Although the 8BP library will provide you with very useful functions in videogames, you will have to use it with caution because every command you invoke will go through the parsing layer of BASIC, before reaching the machine code underworld where the function is located, so the performance will never be optimal. You will have to be clever and save instructions, measure the execution times of instructions and chunks of your program and think of strategies to save execution time. It's an adventure of ingenuity and fun. Here you will learn how to do it and I will even introduce you to a technique I have called "massive logics" that will allow you to speed up your games to limits you might have thought impossible.

In addition to the library, you have at your disposal a simple but complete sprite and graphics editor and a series of magnificent tools that will allow you to enjoy the adventure of programming a microcomputer in the 21st century.

The following "nice" drawing schematized the capabilities that 8BP provided and was used in a "retro" fair. Nowadays it has more capabilities.

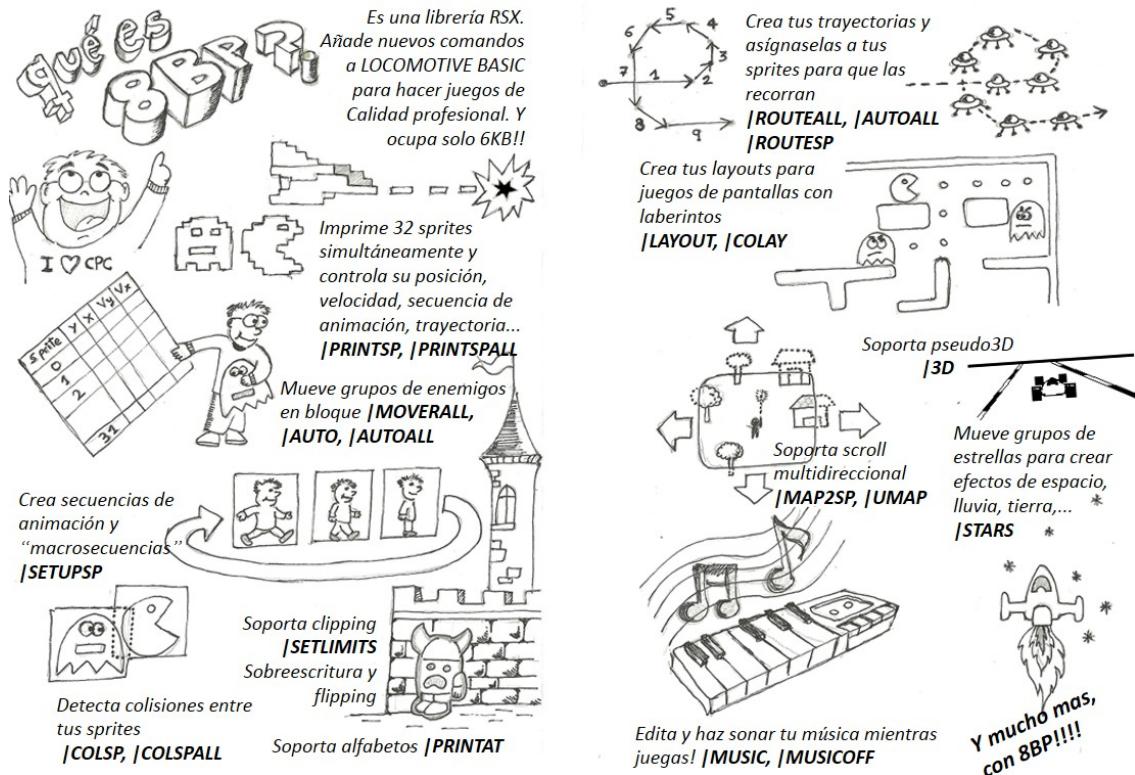


Fig. 3 Summary of 8BP (currently has more commands)

## 2.1 What is an RSX library?

RSX is the acronym for Resident System eXtensions. Libraries such as 8BP that provide commands to extend BASIC are called RSX libraries.

On the CPC6128, some of the commands used to drive the disk drive are pre-installed RSX commands, such as |TAPE, |DISC, |A, |B, |CPM and others. If this functionality did not exist, each 8BP routine would have to be invoked with a CALL <address>, so the existence of RSX makes the programs more understandable.

Not everything is peace and harmony. Using RSX is slower than using CALL directly and also if we declare 10 new commands in a library, the tenth command may take 1ms longer to start executing than the first one. The 8BP library has 27 commands and the last one starts executing 2ms later because it is at the bottom of the list. This is one of the problems of being under the BASIC interpreter.

8BP compensates for this problem by putting the most frequently used commands at the top of the list, and leaving the less frequently used commands at the bottom. As you will soon deduce, the most frequently used command in 8BP is |PRINTSPALL, which prints all sprites on the screen. This command is therefore at the top of the list.

## 2.2 Functions of 8BP

After loading the library with the command: LOAD "8BP.BIN" and invoke from BASIC the \_INSTALL\_RSX function (defined in machine code) using the BASIC command:

**CALL &6b78**

You will have the following commands at your disposal, which you will learn to use with this book

3D, <flag>, #, offsety  3D, 0	Activates the pseudo 3D projection mode.
ANIMA, #	Changes the frame of a sprite according to its sequence
ANIMALL	Changes the frame of sprites with animation flag activated (no need to invoke it, a flag in the PRINTSPALL instruction is enough to invoke it).
AUTO, #	Automatic movement of a sprite according to its Vy,Vx
AUTOALL, <flag routed>, <flag routed>, <flag routed>, <flag routed>.	Movement of all sprites with automatic move flag on
COLAY, threshold_ascii, @collision, #  COLAY, @collision, #  COLAY, #  COLAY	Detects collision with the layout and returns 1 if there is collision. Accepts a variable number of parameters (always in the same order) from 4 to none.
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%.  COLSP, 32, ini, end  COLSP, 33, @collided%  COLSP, 34, dy, dx  COLSP, #	Returns first sprite with which # collides. The command can be configured with codes 32,33 and 34.
COLSPALL,@who%, @withwho%.  COLSPALL, collider  COLSPALL	Returns who collided (collider) and with whom it collided (collided).
LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$.	Prints 8x8 image strip and fills map layout
LOCATESP, #, y, x	Changes the coordinates of a sprite (without printing it)
MAP2SP, y, x  MAP2SP, status	Creates sprites to paint the world in scrolling games. Sprites are created with state = status
MOVER, #, dy, dx	relative movement of a single sprite
MOVERALL, dy,dx  MOVERALL	Relative motion of all sprites with relative motion flag active
MUSIC, C, flag, song, speed  MUSIC, flag, song, speed  MUSIC	A melody starts to play. Channel C can be turned off for use with FX effects if desired No parameters stops sounding.
PEEK, dir, @variable%	Reads a 16bit data (can be negative)
POKE, dir, value	enter a 16bit data (which can be negative)
PRINTAT, flag, y, x, @string	Prints a string of redefinable "mini-characters".
PRINTSP, #, y, x  PRINTSP, #  PRINTSP,32, bits	prints a single sprite (# is its number) regardless of status byte. If 32 is specified then we set background bits
PRINTSPALL, ini, fin, anima, sync  PRINTSPALL, ordermode  PRINTSPALL	Prints all sprites with active print flag. If invoked with a single parameter, the ordering mode is set.
RINK,tini,colour1,colour2,...,colourN  RINK, jump	Rotates a set of inks according to a definable pattern made up of any number of inks
ROUTESP, #, steps	Makes you go through N steps of the sprite route at once.
ROUTEALL	Modify sprite speed with path flag (no need to invoke it, just flag in AUTOALL).
SETLIMITS, xmin, xmax, ymin, ymax	Defines the game window, where clipping is done.

SETUPSP, #, param_number, value	Modifies a parameter of a sprite. If parameter 5 is specified, Vx can optionally be supplied.
STARS, initstar, num, colour, dy, dx	Scroll of a set of stars
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Updates items on the world map with a subset of items from a larger map

*Table 2 Commands available in the 8BP library*

Note that a vertical bar appears at the beginning of each one because they are BASIC "extensions". Some variables appear with the symbol "%" to denote that they are integers (not decimals) but if you use DEFINT to force all variables to be integers you don't need the "%".

In addition, you have an experimental command:

#### **|RETROTIME, date**

This command allows you to transform your CPC into a time machine, just by entering the desired target date. The only limitation of the command is that you must enter a date equal to or after the date of birth of the AMSTRAD CPC, April 1984,

#### **|RETROTIME, "01/04/1984".**

Please use this feature with caution. You could create a time paradox and destroy the world.

Although you may be sceptical at the moment about what you can do with the 8BP library, you will soon discover that using this library together with the advanced programming techniques you will learn in this book will allow you to make professional games in BASIC, something you might have thought impossible.

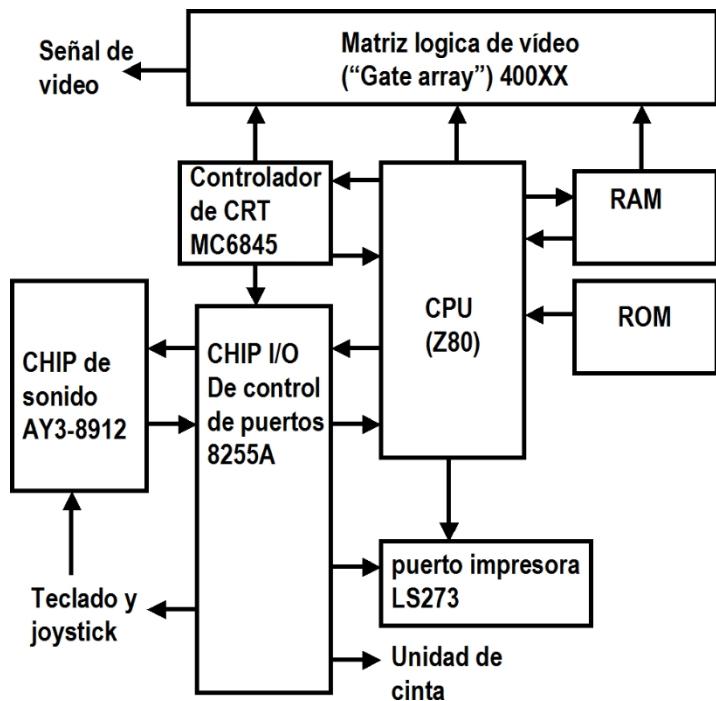
#### **Important note for the programmer:**

The 8BP library is optimised to be very fast. That is why it does not check that you have set the parameters of each command correctly, nor that they have the right value. If any parameter is wrongly set, it is very likely that the computer will hang when executing the command. Checking these things takes execution time and time is a resource that cannot be wasted, not even a millisecond.

## **2.3 AMSTRAD CPC Architecture**

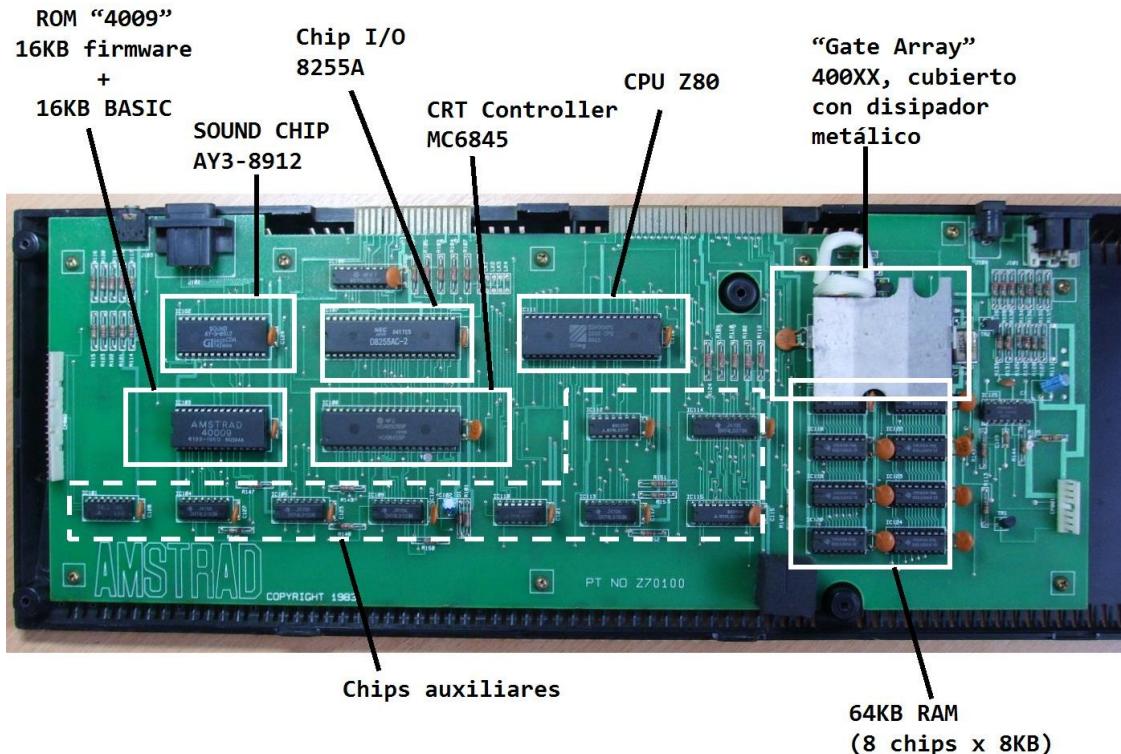
This section is useful to understand later how the 8BP library uses memory.

The AMSTRAD is a computer based on the Z80 microprocessor, running at 4MHz. As can be seen in its architecture diagram, both the CPU and the video logic array (called "gate array") access RAM, so in order to "take turns", memory accesses from the CPU are delayed, resulting in an effective speed of 3.3Mhz. This is still quite a lot of power.



*Fig. 4 Architecture of the AMSTRAD*

The video RAM is accessed by the gate array chip 50 times per second in order to send an image to the screen. In older computers (such as the Sinclair ZX81) this task was entrusted to the processor, taking even more power away from it.



*Fig. 5 Identification of components on the board*

The gate array is a chip containing many logic gates, designed specifically for AMSTRAD. In ZX Spectrum there is a similar chip called ULA (Uncommitted Logic Array - not to be confused with ALU). Both the amstrad and the ZX are chips designed exclusively for these computers. On the ZX, as well as being used to generate

the video image was also used to read the keyboard and cassette input, however, on the AMSTRAD these functions are performed by other chips such as the 8255A. The Z80 has a 16bit address bus, so it is not capable of addressing more than 64KB. However, the Amstrad has 64kB RAM and 32kB ROM. In order to address them, the AMSTRAD is able to "switch" between banks, so that, for example, if a BASIC command is invoked, it switches to the ROM bank where the BASIC interpreter is stored, which overlaps with the 16KB of screen space. This mechanism is simple and effective.

In addition to the ROM containing the 16KB BASIC interpreter located in the high memory area, there is another 16KB of ROM located in the low memory, where the firmware routines (what could be considered the operating system of this machine) are located. In total (BASIC interpreter and firmware) they add up to 32KB.

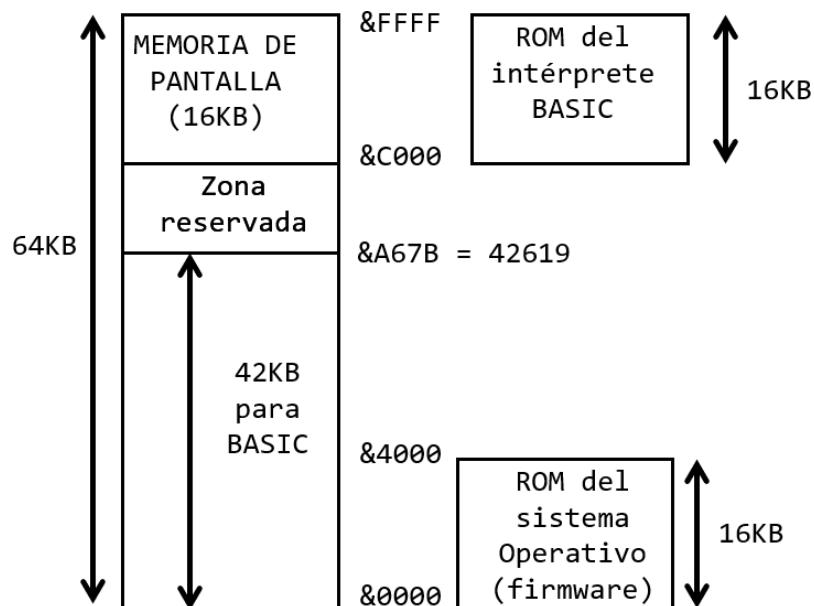


Fig. 6 AMSTRAD Memory

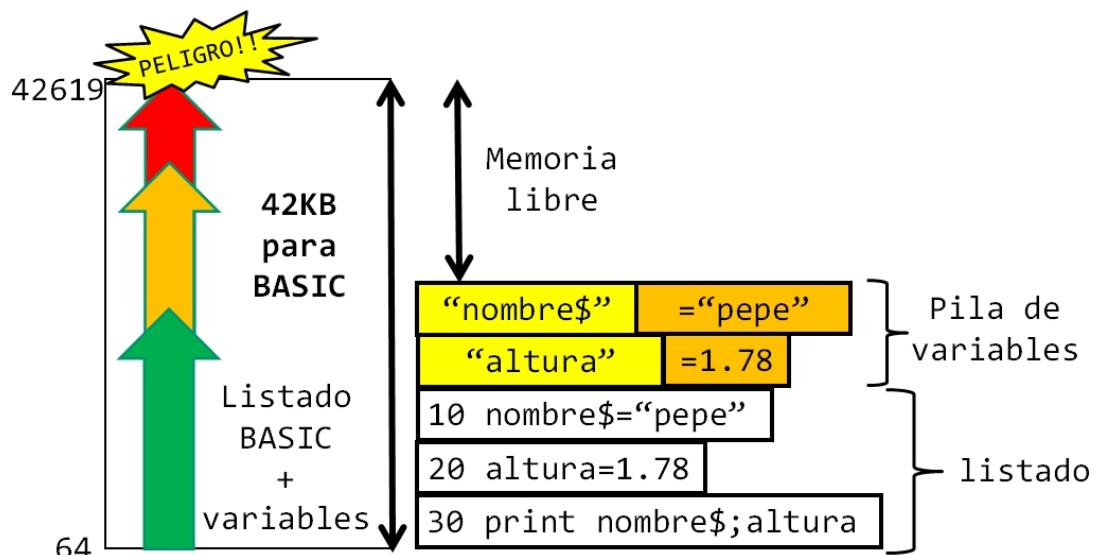
As can be seen in the memory map, of the 64KB of RAM, 16KB (from &C000 to &FFFF) is video memory. BASIC programs can occupy from position &40 (address 64) to 42619, because beyond that there are system variables. This means that about 42KB are available for BASIC, as we can see by printing the system variable HIMEM (abbreviation of "High Memory").

```
print HIMEM
42619
Ready
```

Fig. 7 HIMEM system variable

The operation of BASIC takes into account the storage of the program in increasing addresses from position &40. Once in execution, the variables that are declared must occupy space to store the values they take and since they cannot occupy the same area where the program is stored, they simply start to be stored above the last address occupied by the BASIC list.

On the AMSTRAD, each variable occupies information with its name and value. A numeric variable of type integer will occupy 2 bytes of memory for the value, but occupies as many bytes to store the name. Real variables (with decimals) take up 5 bytes. Each time we use a variable we will consume memory starting at the first free address above the BASIC listing. **There is a danger that if we create many variables and our BASIC list is very long, the stack of variables may eat up all the free memory.** In that case, the BASIC program would be corrupted and would stop working. As soon as the program starts executing, the variables will start eating space and when they fill all the available RAM, it will give a memory full error.



*Fig. 8 Growth of memory consumption of BASIC lists and variables*

You can monitor at any time how much memory you have available with the **FRE(0)** command, whose value you can print or load into a variable. You can do this experiment, you will see how you manage to consume all the memory of the CPC with an array of integers:

```
print fre(0)
42209
Ready
1st
10 CLEAR:DEFINT a-z
20 DIM a(21099)
30 PRINT FRE(0)
Ready
run
0
Ready
```

This simple program occupies all the memory between the sum of the BASIC list and the array.

Notice how FRE(0) tells us that there is not a single byte left.

Each time a value is assigned to a literal variable, such as name\$, the variable will be relocated, leaving less and less space available, although the space it previously occupied will be marked as "available". When a program runs out of memory, it will compact all variables, freeing all "available" space. This effect is also achieved by executing **FRE("")**. But be careful, the Amstrad BASIC does not allow you to simply execute **FRE("")**, because it will give you a "syntax error". You must at least assign a variable, for example **p=FRE("")**. This command performs what is nowadays known in some languages as "garbage collection".

If during the execution of a program you have a memory full problem, you may be able to solve it simply by periodically executing a line like this:

```
100 c=c+1
110 if c and 63 =0 then 120 else p=fre("") :FRE every 64 cycles
120 the programme continues
```

This solution works only if the problem is due to some instruction that consumes a bit more than what is free just before executing the FRE automatically. If that is the case, this solution will work. However, it is very "rare" to solve it this way, because if the Amstrad has no memory, it does FRE automatically and in theory it doesn't need to do it. Another simpler (and more effective) solution is to delete or shorten REM lines, to give a little more free memory to the Amstrad. If there is enough memory and you still have a memory full, it is a problem of too much GOSUB without RETURN.

#### **Each numeric variable consumes the following memory:**

- the variable name: N bytes
- the value, with the last byte set to zero indicating end of literal: 2 bytes

#### **Each variable literal consumes the following memory:**

- the variable name: N bytes
- memory address (or "pointer") where its value starts: 2 bytes
- the value, with the last byte set to zero indicating end of literal: N bytes

Whenever a literal variable is relocated by assigning it a new value, the new value is loaded into the available memory area and the pointer is reassigned to point to the new address, leaving the old value unpointed by any variable. That previous space or "gap" is available, but a cleanup is needed to compact all variables and free all available gaps.

Let's see a simple example that shows you the available space while relocating a literal (or "string") variable, apparently spending more and more memory, although the unused memory remains available (they are "holes" in the variable stack). If you try to do the same with a numeric variable, the memory consumption will be constant because numeric variables always occupy the same and do not relocate when changing value, while literal variables (or "strings") change length when changing value.

```

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

```

As you can see in each iteration there is less memory available, but if you let it run indefinitely, when it runs out of memory AMTRAD will execute a procedure to clean up unused memory and it will have memory again. That procedure is the same as when you run FRE(" ").

Not to be confused with the CLEAR command which frees space by removing all variables from the variable stack.

It is advisable to run FRE(" ") periodically in your program, since the operation of compacting all the variables when all the memory has been consumed is more work (and therefore slower) than if you periodically do a FRE(" ") that has little work to do.

If your program after some time running produces a memory full error, try to run a FRE("") periodically, delete "REM" lines to give it more free memory or check that it is not an excess.

nesting of GOSUB, which is the most frequent error.

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

<after many iterations>

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

Finally a curiosity: the CPC 464 gives you a FRE(0) of 43,553 bytes free while the 6128 gives you less memory, specifically 42,249 bytes. This is directly related to the fact that if you do a PRINT HIMEM on the CPC464 you get 43903, while on the CPC6128 you get 42619.

### 2.3.1 GOSUB /RETURN Stack

Every time you run GOSUB in the Amstrad BASIC, the system must point to where it has to return to when you RETURN. Well, the Amstrad CPC has a stack of 83 positions to store the jumps. It is common to make a mistake in programming and in some IF of the subroutine return with a GOTO (i.e. not return) so they accumulate if you are not careful and you can get a "memory full" error during the execution of your program.

<pre> <b>10 GOSUB 100</b> <b>20 REM here never arrives</b> <b>100 i=i+1</b> <b>110 PRINT i</b> <b>120 GOTO 10</b> </pre>	<pre> 73 74 75 76 77 78 79 80 81 82 83 <b>Memory full in 100</b> <b>Ready</b> </pre>	
--	--	--

In this example, even if you print the remaining memory by replacing line 110 with :

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

You will see that the available memory does not go down and yet, even though you have almost all the memory free, the Amstrad gives memory full. This case is due to **GOSUB** nesting, not because there is no free memory. **Only 83 GOSUB are allowed without RETURN.**

I can't tell you for sure which is the memory area where the Amstrad BASIC interpreter stores the 83 addresses for RETURN, but it is probably in the system memory area (6KB from 42619 to 49152). It is a 6KB area just before the screen memory (49152 is &C0000) where BASIC stores the rewritable characters, and also the "stack". This is an area used to store the memory address where a program is located before jumping to a routine, so that it can return to where it was when the routine ends. It is used in low level (assembly language). The stack grows to lower addresses as it stores addresses (i.e. it starts at 49152 and picks up smaller and smaller addresses) and when returning from a routine it decreases again. You might think that, if one routine calls another routine and that routine calls another routine and so on, the stack would grow so much that it would leave the "system" area and invade other areas, but don't worry, 8BP keeps the stack under control and so does the Amstrad BASIC.

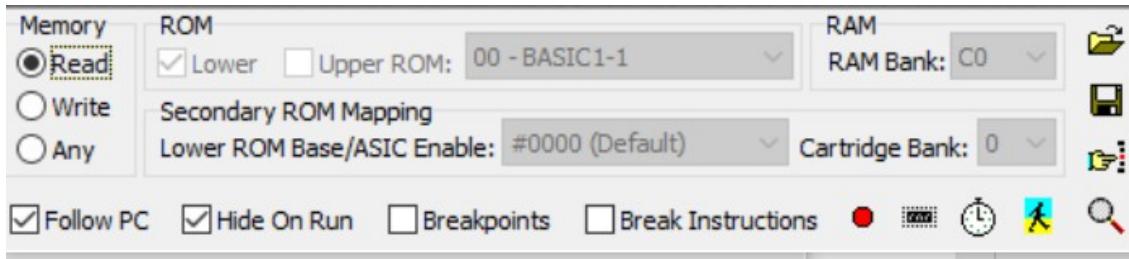
### 2.3.2 An experiment to see the ROM with Winape

You can make use of Winape to see what is contained in the BASIC interpreter ROM that overlaps with the screen addresses, and the same with the operating system ROM.

From the emulation window type

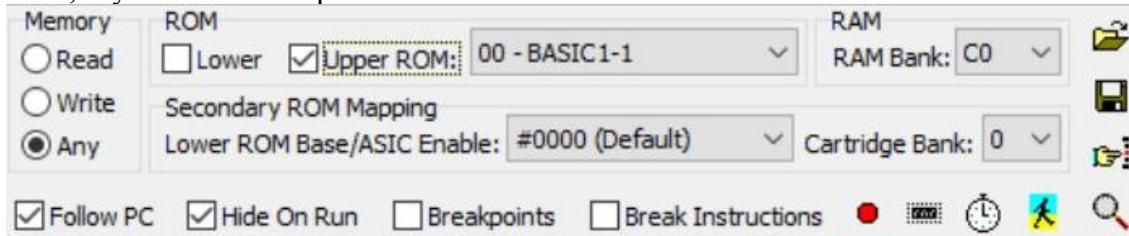
**POKE &C000, &FF**

Then press pause on the emulator and you will see a screen with memory addresses and values. Look for address &C000 and its contents. It should be &FF. At the bottom you will see something like:



*Fig. 9 Winape showing RAM*

Well, if you now set it up like this:



*Fig. 10 Winape showing ROM*

You will see the contents of address &C000 in the ROM bank containing the BASIC interpreter. It is very interesting to be able to do this with Winape.

## 2.4 8BP Memory Map and Assembly Options

Since the Amstrad BASIC starts by consuming the lowest addresses, in order for them to "coexist", the 8BP library is loaded into the high available memory area. It is important to understand how BASIC works in order to use the library as you will have to use a BASIC "MEMORY" command.

```
10 a=5
20 k=@a
30 PRINT k
Ready
run
411
Ready
```

The text of a program written in BASIC is stored starting at address &40 (in decimal it is 64, a very "low" address) and the variables that your program creates are stored occupying positions just after the memory occupied by the list. The following example shows where the variable "A" is stored, which happens to be address 441.

The BASIC list and variables can grow very large if your program is very large, and could invade the memory area where the 8BP routines are stored. To avoid that, you must execute a **MEMORY** instruction that protects the memory area where 8BP, your graphics and your music are stored. The **MEMORY** command is like a "barrier" that prevents BASIC and its variables from occupying memory beyond the limit we tell it to. If it does, it will give a "**MEMORY FULL**" error and stop working.

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



### Programa BASIC y variables

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

The 8BP library leaves you between 24 and 25 KB free for your BASIC listing or your C compiled program (in 8BP you can also program in C). From version V42 of 8BP it is possible to choose several **assembly options depending** on the type of game you are going to develop. The idea is simple: if you are going to make a game with scrolling you need the commands that deal with scrolling, but you don't need the commands that deal with drawing mazes (LAYOUT in 8BP) or detecting collisions with the maze walls. That is, from V42 onwards, some 8BP functions **will not use memory if they are not needed**. So 8BP can leave you more free memory for your BASIC listing or your C program. You will have to choose an **assembly option** depending on the type of game you want to program. This table summarises the available options. You will soon understand the SAVE command that appears in each description. This table is also available in an appendix

Option	Description of the option	Example of a typical game
0	You can do any game All commands available You need to use <b>MEMORY 23499</b> To save library + graphics + music: <b>SAVE "8BP0.bin",b,23500,19119</b>	anyone
1	maze or screen passing games You need to use <b>MEMORY 24999</b> <b>Not available in this mode:</b> <b> MAP2SP,  UMAP,  3D</b> To save library + graphics + music: <b>SAVE "8BP1.bin",b,25000,17619</b>	
2	For scrolling games You need to use <b>MEMORY 24799</b> Not available in this mode: <b> LAYOUT,  COLAY,  3D</b> To save library + graphics + music: <b>SAVE "8BP2.bin", b,24800,17819</b>	
3	For games with pseudo 3D You need to use <b>MEMORY 23999</b> Not available in this mode: <b> LAYOUT,  COLAY</b> To save library + graphics + music: <b>SAVE "8BP3.bin", b,24000,18619</b>	

As you can see the option that leaves you the most free memory for your program is option 1, as it leaves you 25KB free for your game. Option 3 leaves you 24KB and option zero leaves you 23.5KB. I don't recommend you to use option zero, unless you really need it. It is better to choose the right option for your game and you will have more memory available.

These 24-25 KB free are for your listing, as the music and graphics of your game are stored in another "higher" area. For example, with option 1 you have:

- 25 KB free for your listing (BASIC or C) and variables
- 1.5 KB free for your music
- 8.5 KB free for your graphics

**TOTAL: 35 KB free for your game**

The assembly option you choose is defined in an 8BP file called "make\_all\_mygame.asm". This file has a line that **you must edit to choose the option you prefer**.

```
; 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"
```

Before loading the library, you must run the **MEMORY** command with the limit associated with the assembly mode you have chosen for your game type. From version V42 there are 24KB, 24.8 KB or 25KB free depending on the assembly option you choose.

Regardless of which assembly option you choose, the memory addresses where all commands are located are exactly the same (Appendix XI). For

For example, the LAYOUT command can be invoked if you choose assembly option 1 or 2, but if you use assembly option 2 , invoking the LAYOUT command will do absolutely nothing, as will the MAP2SP command if you use assembly option 1.

```

AMSTRAD CPC464 MEMORY MAP of 8BP

;
; &FFFF +-----+
; | display + 8 hidden segments of 48bytes each
; &C000 +-----+
; | system (redefineable symbols, stack pointer, etc.)
; 42619 +-----+
; | bank of 40 stars (from 42540 to 42619 = 80bytes)
; 42540 +-----+
; | character layout map (25x20 =500 bytes)
; | and world map (up to 82 items fit in 500 bytes)
; | Both are stored in the same memory area.
; | because you either use one or you use the other.
; 42040 +-----+
; | sprites (almost 8.5KB for
; drawings) you have 8440 bytes if there are no sequences and no
; | routes)
; +-----+ alphabet images are also stored here.
; | route definitions (of variable length each)
; +-----+
; | animation sequences of 8 frames (16 bytes each)
; | and groups of animation sequences (macro-sequences)
; 33600 +-----+
; | songs
; | (1500 Bytes for music edited with WYZtracker 2.0.1.0)
; 32100 +-----+
; | 8BP routines (8100 bytes or 7100 bytes)
; | here are all the routines and the sprite table
; | includes music player "wyz" 2.0.1.0
; 25000 +-----+
; | YOUR BASIC or CLIST
; | 24KB, 24.8 KB or up to 25KB free for BASIC or C
; | depending on which assembly option you use for 8BP
; |
; 0 +-----+

```

*Fig. 11 Memory using 8BP*

If you run out of space for graphics or music and need more, 8BP allows you to place images and music in other areas (below address 24000) and it will work just fine.

### 3 Necessary tools

**Winape:** emulator for windows OS with editor to edit and test your BASIC program. And also for assembling graphics and music.

**SPEDIT:** ("Simple Sprite Editor") BASIC tool to edit your graphics. The output of spedit is assembler code that is sent to the Amstrad CPC printer. By running the tool inside Winape, the printer is redirected to a text file so that your graphics will be stored in a txt file. This tool has been created to complement the 8BP library and the graphics of all the games I have created have been made with SPEDIT.

**Wyztracker:** for composing music, under windows. The program capable of playing the melodies composed by Wyztracker is Wyzplayer, which is integrated into 8BP. After assembling the music you can make it play with a simple command |MUSIC

**8BP library:** install new commands accessible from BASIC for your program. As you will see, this will be the "heart" that drives the machinery you build.

**CPCDiskXP :** allows you to record a 3.5" floppy disk which you can then insert into your CPC6128 if you have a cable to connect a floppy disk drive. If you want to make a CPC464 audio tape, this tool is not necessary.

**2CDT:** essential tool for creating .cdt files. I usually extract the files from a .dsk to windows disk using CPCDiskXP and then use 2cdt to create the cdt file.

**Tape2wav:** tool to create .wav files from .cdt files

#### OPTIONALLY:

**ConvImgCPC:** upload image editor for your games. Also converts from BMP. Programmed by Ludovic Deplanque ("DEMONIAK")

**RGAS:** (Retro Game Asset Studio) powerful sprite editor, evolved from the AMSprite tool, created by Lachlan Keown. This sprite editor is 8BP compatible and runs under Windows. When you outgrow Spedit, this may be the best option.

#### NOT RECOMMENDED:

**Fabacom:** compiler executable inside the AMSTRAD CPC 6128 or from the Winape emulator to compile your BASIC program and make it run faster. It is compatible with the 8BP library command calls. However, it is not recommended for several reasons:

- Your program will take up much more space because fabacom needs an additional 10KB for its libraries, and also, once it compiles your program it still takes up the same amount of space.

so that a 10KB BASIC program is transformed into a 20KB BASIC program.

- There are some documented incompatibility problems of this compiler with some BASIC instructions.
- Moreover, as you will see throughout this book, you can achieve very high speed without compiling.

**CPC BASIC compiler:** executable compiler for windows. It is compatible with the 8BP library command calls. Unlike fabacom, the compiled program only takes up about 5KB extra, however, it reserves 16KB of work to run so that it hardly leaves any space for your program, since it "steals" 20 KB in total. And it is not 100% locomotive BASIC compatible.

The speed gain with both fabacom and CPC Basic compiler can be up to 50%, depending on the game. That is, a game running at 20 FPS would run at 30 FPS. This is not bad, but think that we have gone from interpreted BASIC to machine code and normally it is said that the speed must be multiplied by at least 100 (we would be talking about an increase of 10000%). However, we have only gained 50%. The reason for such a "poor" gain is that the 8BP instructions already do all the hard work and in fact the compiler only translates into machine code the less heavy part, the game logic.

Finally, you should know that, if you want your programs to run much faster, since v40 of 8BP there is C language support, so you can either program your whole game directly in C or program in BASIC and translate only the "game cycle" to C. There is a chapter in this manual dedicated exclusively to this topic. There is a chapter in this manual devoted exclusively to this topic.

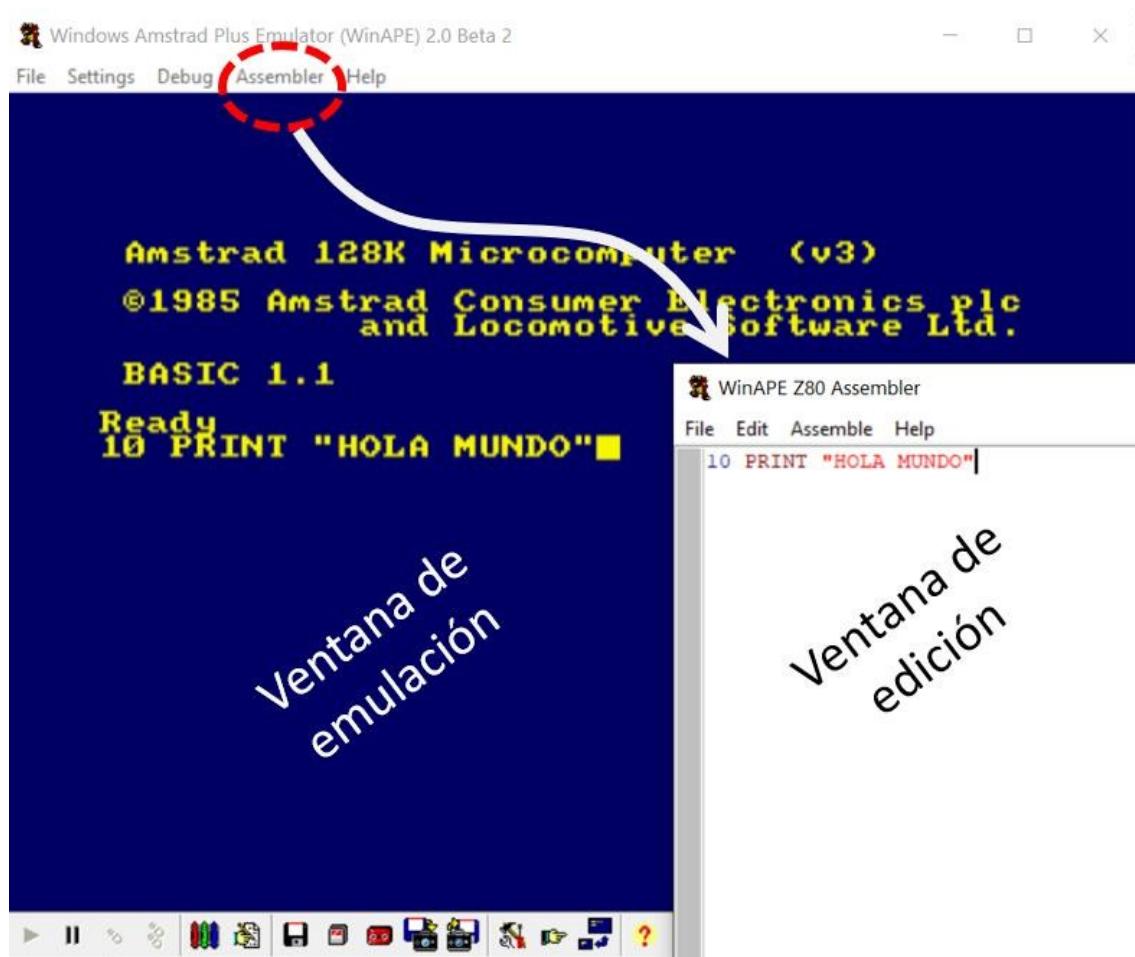
## 4 First steps with 8BP

### 4.1 Install winape

The first thing to do is to install the latest version of **Winape**, which is an Amstrad emulator, editor and assembler. You can download it from [www.winape.net](http://www.winape.net)

### 4.2 Getting familiar with winape: "hello world".

Once Winape is installed, familiarise yourself with it by trying out some Amstrad games and trying to change the configuration. Try opening the built-in assembler menu and edit a "hello world" in the assembler window. Then copy and paste the text into the emulation window. You will see how it is pasted character by character

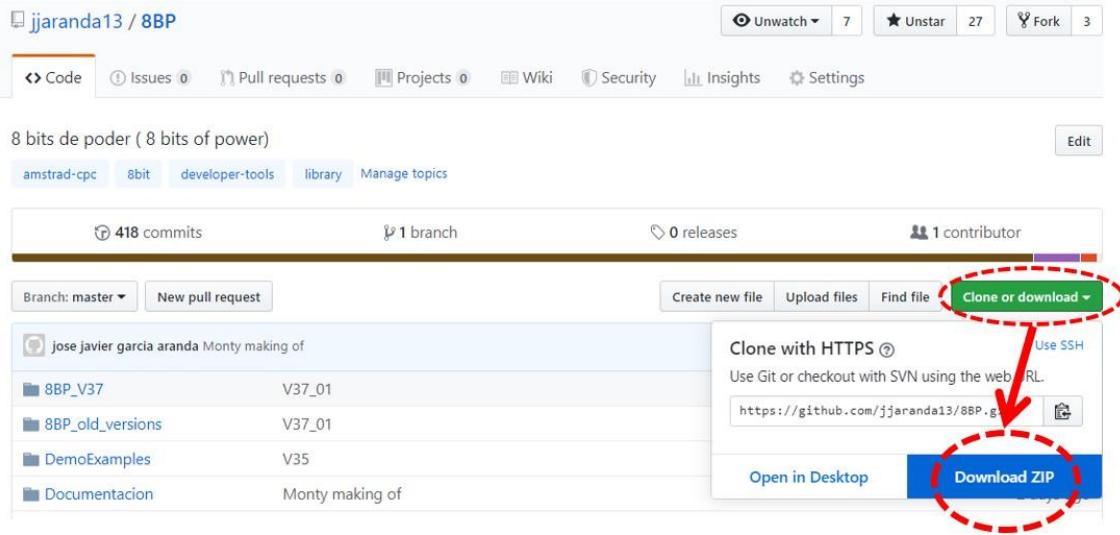


If you make a longer program, to copy it to the emulation window it is very interesting to use the settings->high speed option. You will see how fast it is copied.

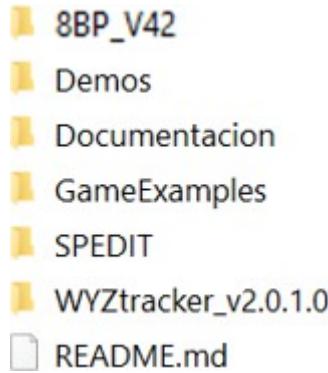
### 4.3 Download the 8BP library

We are getting to the interesting part. Go to the url <https://github.com/jjaranda13/8BP> and download the zip (<https://github.com/jjaranda13/8BP/archive/master.zip>) . To do this

you can simply click on the green "clone or download" button and then select the zip file



Once you have downloaded it, unzip it in the directory of your choice. You will get the following result:



#### 4.4 Run the demos

Now we are going to make a first contact with 8BP by watching some demos. Go to the Demos directory. In it you will find a series of subfolders: **ASM, BASIC, C, DSK, MUSIC, ASM, BASIC, C, DSK**.

Go to DSK . There you will see a .dsk file with the demos. From winape go to the File-menu.

>drive A-> insert Disc image and select the demo file

Once selected, from the Amstrad emulation window, type **CAT** to see the files.

```
cat
```

```
Drive A: user 0
```

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

```
89K free
```

```
Ready
```

Each .BAS file is a demo where you can see some of the features of 8BP (not all features can be seen in the demos, but there are some representative ones).

Execute the **RUN** command "**LOADER.BAS**". You will get the following menu:



You can now choose a demo and try it out. Enjoy. In the next and last step we will start the creation of a game.

#### 4.5 Creating your first programme with 8BP

We have tested a .dsk file containing many demos, with graphics and music. The **Demos/ASM** directory and the **Demos/MUSIC** directory contain the graphics and music of the demos you have tested. However, if you want to make your own game or demo, it is better to start with "clean" files without all the graphics required by the demos you have tested.

We go to the root directory. There you will find a folder called "**8BP\_V42**". I recommend that you make a copy of this folder and rename it to "**my\_game**". That way you will preserve the original "**8BP\_V42**" folder, even if you start changing things.



Inside the folder "**8BP\_V42**" you will find the folders ASM, BASIC, DSK, MUSIC, TAPE, and output\_spedit.

From the winape assembler window open the file "ASM/make\_all\_mygame.asm" and run it (simply in the winape z80 assembler menu select "Assemble" or press Ctrl+F9). This will start assembling (copying in memory) the library and the graphics into the Amstrad memory. In this case we are going to assemble very few graphics, only the essential ones for a small game. The graphics are in the file "**images\_mygame.asm**".

After assembling everything, you will get a message like the one below:

```

; 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
; disponiblos los comandos |LAYOUT|
; ASSEMBLING_OPTION = 2 --> para juegos con scroll, MEMORY 2
; disponiblos los comandos |MAP2SP|
; ASSEMBLING_OPTION = 3 --> para juegos pseudo 3D , MEMORY 2
; disponiblos los comandos |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"

```

Assembling : E:\...\PROYECTO\_V42\8BP\_V42\ASM\make\_all\_mygame.asm  
Output To : Direct to Emulator  
Pass : 2 Current Line : 36 Total Lines : 11396 Errors : 0

000059 A43D	;-----
000060 A43D	; FASE 1
000061 A43D	;-----
000062 A43D	;dw 36,0,IMAGE0; 1
000063 A43D	;dw 36,20,IMAGE0; 2
000064 A43D	;dw 36,24,IMAGE0; 3
000065 A43D	;...etc
000070 A43D	_END_MAP
000027 A43D	read "map_table_mygame.asm"
000033 A43D	read "make_graficos_mygame.asm"

OK

C:\...\PROYECTO\_V42\8BP\_V42\ASM\Make\_all\_mygame\8bitsdepoder\_v042\_000\Images\_mygame\Fre\_ejemplo\Tu\_primer\_juego\Sequences\_mygame\Loader\Loader\Tu\_primer\_juego\_c\

Press "ok" and from the assembler window we are going to open another file. In this case we are going to open a BASIC file, specifically "**your\_first\_game.bas**" which is located in the BASIC folder. After opening it, you will see the following list on the screen, which contains 32 lines:

```

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 imprimirlo 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

```

Select everything and copy it. Then go to the CPC emulation window and paste it using the **FILE->paste menu**.

As the list is a bit longer than "hello world", use the winape menu and select **settings->high speed** to copy it and then go back to "normal speed". As the library and graphics are already assembled, you can RUN and the game will run. You must dodge balls falling from the sky to avoid dying, moving left and right.



You can try modifying the program and see its effects. Little by little you will learn 8BP and you will be able to make interesting modifications such as changing the frequency with which enemy balls come out or their speed, or replacing the soldier with a spaceship and the balls with enemy ships with different trajectories.

If you want to test how fast it works in C language, just load the .dsk and run "loader.bas", it will load a version of the game that allows you to choose between the BASIC version and the C version so you can compare. There is a chapter in this book dedicated to C programming using 8BP and a mini BASIC so you can program in C just as you would in BASIC. If you don't have much experience in C programming, I recommend you to go slowly and program in BASIC, the results will be fast and professional.

## **4.6 Create your .dsk with your 8BP set**

Finally, we are going to create a disc with your game. To do this, after having the game running, you must follow these steps

- Create a new disc via winape: FILE-> drive A-> new Blanc Disc
- Format it: FILE->drive A->Format Disc Image
- After you have created your dsk file, from the emulation window, run the following commands:

**SAVE "8BP0.bin", b, 23499,19119**

**SAVE "juego.bas"**

The first command saves the 8BP library, graphics and music to disk. In this case we use assembly option 0 (see options in previous chapter) but you could use any option for this example perfectly well. I have named the file "**8BP0**" to somehow reflect that assembly option 0 has been used, but the name of this binary can be anything.

We are almost done. Now you must select another disk from the winape menu or exit winape so that the .dsk you have created becomes a reality in the windows file system.

Exit winape and reopen it.

Select the disc you have created and run:

```
MEMORY 23499
LOAD "8BP0.BIN"
RUN "game.bas" RUN "game.bas" RUN "game.bas" RUN "game.bas" RUN
```

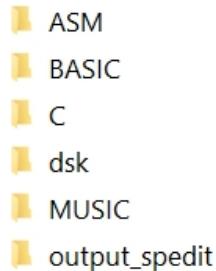
**Hallelujah!**

## 5 Steps to take to make a game

### 5.1 Directory structure of your project

When programming your game, it is best to structure the different files in 7 folders, depending on the type of file they are.

It is perfectly possible to put everything in the same directory and work without folders, but it is "cleaner" to do it the way I am going to present it below.



*Fig. 12 Directory structure*

- **ASM:** here you will put text files written in assembler (.asm), such as the 8BP library itself, the sprites generated with the SPEDIT sprite editor, and some auxiliary files.
- **BASIC:** here you will put your game and utilities such as SPEDIT and Loader.
- **C:** this folder is for "advanced" users who want to program in C the whole game or at least the game cycle. It is only necessary if you are going to program some part of your game in C language.
- **Dsk:** here you will put the .dsk file ready to run on an Amstrad CPC. Inside you will have to place 5 files that we will talk about in the following section
- **Music:** with the WYZtracker music sequencer, you can create your songs and store them in .wyz format in this directory. Once you "export" them, an .asm file will be generated that you will have to save in the ASM folder and a binary file that you will also store in the ASM folder (you can also leave them in this folder, as long as you reference them properly in the make\_musica.asm file in the ASM directory).
- **Output\_spedit:** in this folder you can store the text file generated by spedit. SPEDIT sends the sprites to the printer in assembler format and the winape emulator can collect the output from the Amstrad printer in a file. Here we will place it
- **Tape:** here you can store the .wav file if you want to make a tape to load into the Amstrad CPC464, or the .cdt file.

## 5.2 Your game in just 3 files

To generate your game you will need a binary file and two BASIC files. You can generate several independent binary files (one with the 8BP library, one with the graphics, one with the music...) but as these memory areas are contiguous, it is better to generate a single binary file.

The binary file contains the library, the music, the graphics, the world map memory area or layout and optionally the star bank. The idea is to store all the binary components together in one file, like this (depending on the assembly option you will use one or the other of these commands). The name of the file is ended with a number indicating the assembly option, but it can be called whatever you want.

```
SAVE "yourgame0.bin",b,2350 19119
      0,
SAVE "yourgame1.bin",b,2500 17619
      0,
SAVE "yourgame2.bin",b,2480 17819
      0,
SAVE "yourgame3.bin",b,2400 18619
      0,
```

The longitude being the final address minus the initial address. The final address including music and graphics, the map and the star bank, is 42619, so the length can be calculated by subtracting the initial address from 42620. For example, in assembly option 1, we have  $42619 - 25000 = 17619$ , just the length shown in that command.

The second file is your basic set

```
SAVE "tujuego.bas"
```

And the third file is the loader ("loader.bas"), which will simply be:

```
10 MEMORY 23499
15 LOAD "!pant.scr",&c000: 'only if your game has loading screen
20 LOAD "yourgame0.bin"
50 RUN " !yourgame.bas"
```

I have placed an initial screen load at the initial screen memory address (&C000). At the end of this manual you will find one of the many ways to make a screen load. It is optional. You can make a game with or without a loading screen.

This 3-file method is useful especially if you occupy almost all the memory available for graphics. On cassette tape games loading 18KB can take a while and if you don't use the 8.5KB of graphics, it might be better to load the different binaries separately to save loading time. For example, if you use only 2KB of graphics, with a single binary of length 18619 you would load 8.5KB of graphics, i.e. 6.5KB extra empty. This in tape load time can mean almost two minutes extra time. On disk (CPC 6128) it doesn't matter because it takes no time at all to load them. In that case it's better to make one or two binaries that store only the memory you use.

To make these 3 files you must follow these steps:

### STEP 1

Edit graphics with SPEDIT and the result (SPEDIT sends it to a .txt file) copy it to images\_mygame.asm

## STEP 2

Edit music with WYZtracker

Modify music\_mygame.asm to include the created musics

The tunes will be assembled one after the other, so that each tune will start at a different memory address depending on the size of the tune.

## STEP 3

Re-assemble the 8BP library, so that the part of the library that selects the tunes (the player wyz) can know in which memory addresses they have been assembled (there are more dependencies, but that's one of them). Once reassembled, you will have to save everything with one of these commands depending on the assembly option you use (the name can be whatever you want, I have put a number at the end to indicate somehow the assembly option):

```
SAVE "yourgame0.bin",b,2350 19119
      0,
SAVE "yourgame1.bin",b,2500 17619
      0,
SAVE "yourgame2.bin",b,2480 17819
      0,
SAVE "yourgame3.bin",b,2400 18619
      0,
```

This will be a version of the library specific to your game. For example, the command **|MUSIC,0,0,0,3,6** will play melody number 3 that you have composed yourself. The melody number 3 can be completely different in another game.

## STEP 4

Program your game, which must first execute the call to install the RSX commands, i.e. **CALL &6b78**. And something very important: don't forget to include the **MEMORY** command at the beginning, to avoid that the running BASIC stores variables above the address where 8BP starts.

```
MEMORY 23499 :rem use this MEMORY for option 0 MEMORY
24999 :rem use this MEMORY for option 1 MEMORY 24799
:rem use this MEMORY for option 2 MEMORY 23999 :rem use
this MEMORY for option 3 MEMORY 23999 :rem use this
MEMORY for option 3
```

Your game can be programmed using the winape editor, which is much more versatile than the AMSTRAD editor and can be used for both assembler (.asm) and BASIC (.bas) editing. The winape editor is sensitive to keywords and changes their colour automatically, making programming easier. After writing a BASIC program you have to copy/paste it into the CPC window of winape. To make it faster, you can activate the "High Speed" option of winape during pasting, so that the pasting process will be immediate.

## STEP 5

Load everything with a loader.bas , which you will have to do in BASIC.

## STEP 6

Create a tape or disc with your game

## 5.3 Create a disc or tape with your game

### 5.3.1 Making a disc

To create a new disc from winape we do the following

File->drive A-> new blank disk

This will bring up a file management window for you to name the new .dsk file.

Once created, you can save files with the SAVE command. To delete a file you use the "JERA" command (short for ERASE), which only exists on the CPC 6128 as part of the "AMSDOS" operating system (this did not exist on the CPC464 as it worked with cassette tape).

|ERA, "game.\*"

And they will be erased

To load the game you need a loader that loads the necessary files one by one. Something like (the MEMORY command depends on the assembly option):

```
10 MEMORY 23499: 'memory command depends on assembly option
15 LOAD "!pant.scr",&c000: 'only if your game has loading screen
20 LOAD "yourgame0.bin"
50 RUN " !yourgame.bas"
```

To save each of the files you must use the SAVE command with the necessary parameters, for example:

```
SAVE "LOADER.BAS"
SAVE "yourgame0.bin",b,23500,
19119 SAVE
"yourgame0.bin",b,23500, 19119
SAVE "yourgame.BAS"
```

If you want to write the .dsk to a 3.5" floppy disk and connect it to an external floppy drive of your AMSTRAD CPC 6128, you will need the easy-to-use CPCDiskXP program. From a .dsk you can burn a 3.5" floppy disk in double density (don't forget to cover the hole of the floppy disk to "trick" the PC).

### 5.3.2 Making a tape with Winape

The most important thing when creating a tape is to save the files on it in the order in which they will be loaded by the computer. A tape is not like a disk on which you can load any stored file, but the files are one after the other, so you must take special care in this point.

If your game loader is like this:

```
10 MEMORY 23499
15 LOAD "screen.scr",&c000 : rem if your game has loading screen
20 LOAD " !yourgame0.bin"
50 RUN " !yourgame.bas"
```

The exclamation "!" is very important so that the Amstrad does not get the message "press play then any key" when executing each LOAD.

First you must save the loader (let's say it is called "**loader.bas**"), then the file "**tujuego.bin**" and finally "**tujuego.BAS**".

To create a ".wav" or from winape

**file->tape->press record**

You will then be presented with a file management menu so that we can decide what to name the file ".wav".

If you are in CPC 6128 mode, then next you must run from BASIC

**|TAPE**

And then

**SPEED WRITE 1**

With this command what we will have done is to tell the AMSTRAD to record at 2000 baud. This will shorten the loading time. If you don't execute this command, the recording will be done at 1000 baud, which is safer but much slower.

**SAVE "LOADER.BAS"**

You will get a message telling you to press rec&play, and then press "ENTER". Then save each and every file:

```
SAVE "yourgame0.bin",b,23500,  
19119 SAVE  
"yourgame0.bin",b,23500, 19119  
SAVE "yourgame.BAS"
```

Finally, we have to do one last operation to make winape close the file.

**file->remove tape**

After doing the "remove tape", the file will acquire its size (if you don't do it, you can see that on your PC's disk the file does not grow and it is because it has not been dumped to disk).

To load the game, if you are on a CPC6128

**|TAPE**  
**RUN ""**

To reuse the disc

**|DISC**

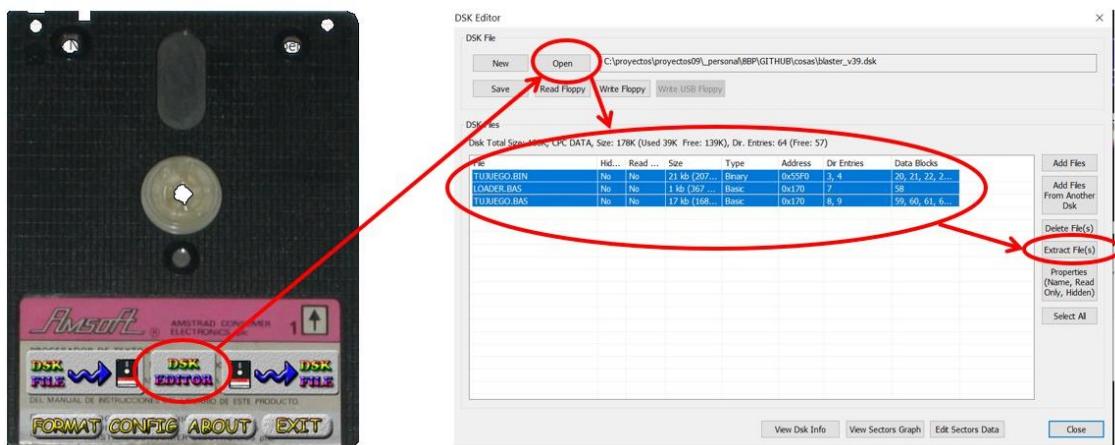
If you want to save a loading screen, see Appendix I on video memory organisation, where I explain how to do it.

**5.3.3 Make a tape easily with CPCDiskXP, 2cdt and tape2wav**

Winape is a great tool for programming and emulation. However it has a small limitation: it doesn't allow you to make a tape in cdt format, only in wav format. There is a very fast and reliable way that will allow you to make .cdt and wav files for the you need the tools CPCDiskXP, 2cdt and tape2wav.

Let's start from the assumption that you have created a .dsk with your files. With that, you need to take the following steps:

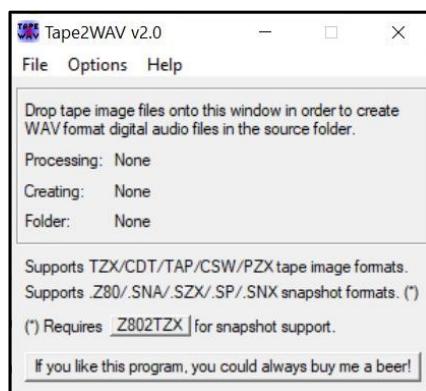
- 1) First of all with the CPCDiskXP tool you can open the .dsk and extract the necessary files for your game (the "loader.bas", the "tujuego.bin" and "tujuego.bas"). To do that you simply click "disk edition", then "open", open your .dsk, select the files and finally click "extract files". Once you have done that, you will have the files in the windows file system.



- 2) Then you use the 2cdt tool to burn the files, one by one to a .cdt file. The commands would be:

```
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
```

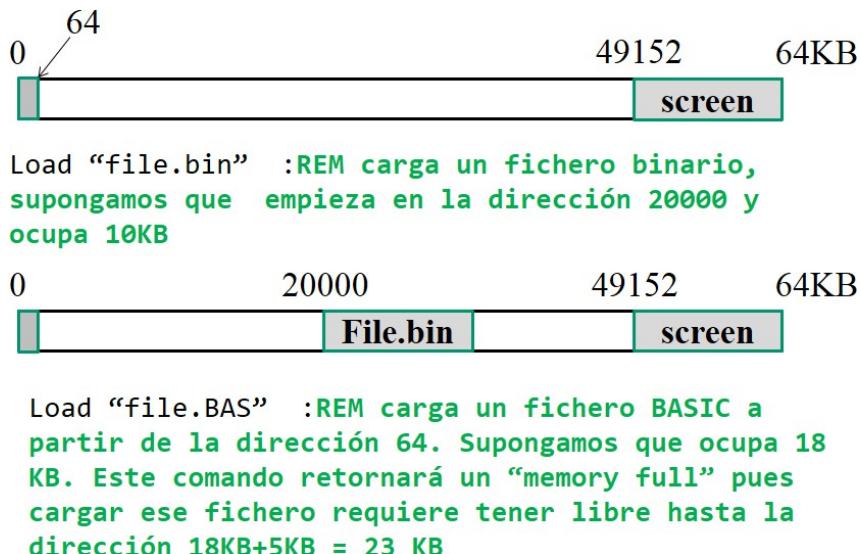
- 3) Now you have the file tucinta.cdt created. If you also want to have a .wav to be able to load it on a real CPC 464, you can make use of the tape2wav tool. Simply start it and drag the .cdt file with the mouse into the tool. The tape2wav will immediately create a file "tucinta.wav" right in the directory where "tucinta.cdt" was.



### 5.3.4 Troubleshooting LOAD and MEMORY

Before loading a BASIC file, the Amstrad makes sure that it will have space available to run it. The BASIC program may only use 1KB of variables, but the Amstrad doesn't know that, so it is more conservative and demands that you have an extra 5KB of empty memory. This 5KB may seem excessive, but the Amstrad does not know a priori how many variables you are going to declare in your program and it would rather have a lot of free space to store variables than to fall short and have the program fail.

This means that if you have previously loaded one or more binary files (with graphics, 8BP library or whatever) and you have left 20 KB free, then you won't be able to load a 20 KB BASIC game but at most a 15 KB game. But don't worry, there are several ways to solve this. I'll explain the easiest one



*Fig. 13 The LOAD problem*

The solution is to make a "loader.bas" file that changes the MEMORY. Let's suppose you have binary data from address 20,000 and your program takes 18KB, leaving less than 5KB of space. All you have to do is:

```

10 MEMORY 19999
20 LOAD "!game.bin": rem loads data from 20000
30 CLEAR: MEMORY 23000 : rem so we give you more free RAM. 40 RUN
"!game.bas": rem the first line of game.bas must be memory 19999

```

This method is very simple and 100% reliable because, although you leave parts of the binary data unprotected during the BASIC load, the first thing you do in BASIC is to execute the MEMORY, so it is protected again before you have created any variables.

Another typical problem related to the MEMORY FULL error occurs when we load a program and in the middle of execution we stop it (by pressing ESC twice). It is possible that we get **MEMORY FULL** when trying to do things like accessing the disk with a CAT command.

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

This is because our BASIC program can consume a lot of RAM memory of variables. Having stopped it does not mean that the variables have disappeared from the program, in fact, they still exist and you can even print them to see their value. What we will do in this case is simply execute the **CLEAR** command, which frees the memory of these variables and then our CAT command (or the one we want).

```
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
```

If you have made a BASIC program so large that you don't have 5KB free between the BASIC program and the assembled binary, then logically you will also get the **MEMORY FULL** error when you go to save your game to disk.

Trying to save the binary will give you a **MEMORY FULL** error, but it's very easy to fix. Just don't load the BASIC listing in your emulator, and don't run any MEMORY command. When you assemble with winape the file "make\_all.asm" you will have all your graphics, music and 8BP library assembled in the Amstrad memory. Then run your SAVE command and it will work. The command to save everything in a single binary depends on the assembly option you put in the **make\_all\_mygame.asm** file, **and will be one of these:**

```
SAVE "yourgame0.bin",b,2350 19119
      0,
SAVE "yourgame1.bin",b,2500 17619
      0,
SAVE "yourgame2.bin",b,2480 17819
      0,
SAVE "yourgame3.bin",b,2400 18619
      0,
```

However, if you have stored binary things (extra graphics, maps, etc.) below the initial address of 8BP you will have to take this into account. For example, if your game starts using memory at address 20000, the command will be (note that I've increased the length so that it takes up from 20000 to 42619)

```
SAVE "yourgame.bin",b,20000, 22619
```

You can now copy and paste your BASIC program into the emulator. Once copied, do not execute any **MEMORY** command, and save your .BAS file to disk.

As you have not yet executed any **MEMORY** command at this point, the Amstrad "thinks" that you have more than 5KB free above your BASIC program and will not give us the **MEMORY FULL** message. Once you execute your **MEMORY** command (for example, if your binary data starts in 20000 it will be a **MEMORY 19999** instead of 23999) the Amstrad will check if you have 5KB free between your BASIC program and the **MEMORY address** and if there is not enough, it will give you an error when executing the **SAVE** command, even if the game works. If during execution your game tries to consume more space than it has free to the **MEMORY address**, it will stop and give a **MEMORY FULL** error.

## 6 Library, music and graphics assembly

This chapter details a bit more what happens when you run the "make\_all" file and will allow you to better understand the whole process, **although if you feel like starting to learn how to program with 8BP, you can skip it and come back here later**, when you want to better understand where to put the graphics and music and how to assemble the library with them.

This is because, for example, the music player is embedded in the library and needs to know where each song starts (memory address), so it is necessary to re-assemble and save the version of the library specific to your game, as well as the assembled graphics file and the assembled music file.

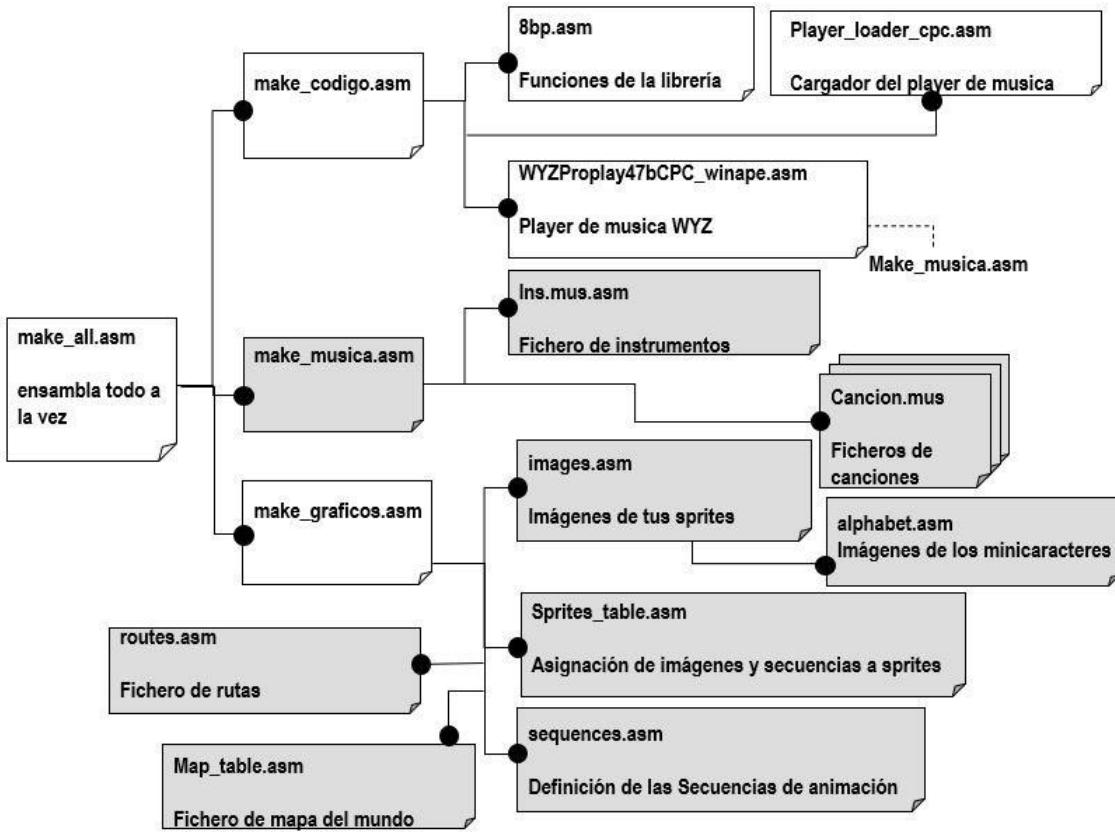
As I explained in the "steps" section, this will be a version of the library specific to your game. For example, the command **|MUSIC,0,0,3,6** will play melody number 3 that you have composed yourself. Tune number 3 may be completely different in another game. The same goes for the data in the instrument file. There are certain dependencies between the music player code and the addresses where the instrument data and melodies are assembled.

It is very simple, but you have to understand the structure of the library to do it, i.e. the structure of the .asm files you have to handle and their dependencies.

The following diagram presents all the .asm files of a game using 8BP as well as the dependencies between them. In the figure, **in grey are those files that you have to edit**, such as:

- the songs and instrument file, which you generate with the WYZtracker
- the file **make\_musica** where you indicate which ".mus" files are to be assembled
- the image file you create with SPEDIT
- the sprite table where you assign images to the sprites (although it is not strictly necessary as you have the **|SETUPSP** command).
- the sequence table, where you define which images make up a sequence,
- the world map: where you define up to 64 elements that make up the world.
- the paths file: where you define the paths of the sprites you want to use.
- the alphabet.asm file: if you want to create an alphabet different from the 8BP standard alphabet

You can assemble everything by opening the file "**make\_all.asm**" and pressing "assemble" in the Winape menu. Then you can use the SAVE command to save the images, music and 8BP library in different binary files, or in a single one, as we have seen.



*Fig. 14 Assembly files*

If you only change the graphics you can assemble them separately, by selecting the file "make\_graphics.asm" and pressing assemble.

If you change the music you must reassemble the library code because there is a dependency between the code and the songs, because the code needs to know where each song starts. So if you change or add songs you must assemble with make\_all.asm . I have reflected this dependency with a dotted line between the player and the make\_musica.asm file.

You may need to assemble something else, like a race track map that uses your images. In that case, add it to the "Make\_graphics.asm" file so that it is assembled after images.asm. The order of assembly is important. First you must assemble the images, associate labels to them, and then you can assemble the maps or tracks that use those labels.

## 6.1 Make\_all.asm

This is the file that allows everything to be assembled. Internally it invokes three files that assemble the library and music player code, the songs and the graphics.

**; Makefile for video games using 8bit power**  
**If you alter a part of it, you only have to**  
**; assemble the corresponding make**  
**for example you can assemble the make\_graphics if you change drawings**  
**SINCE V42 THERE ARE ASSEMBLY "OPTIONS".**

**;** -----

```

; ASSEMBLING_OPTION = 0 --> all available commands.

; ASSEMBLING_OPTION = 1 --> for maze games. MEMORY 25000
;                               available the commands |LAYOUT, |COLAY
;
; ASSEMBLING_OPTION = 2 --> for scrolling games, MEMORY 24800
;                               available the commands |MAP2SP, |UMA
;
; ASSEMBLING_OPTION = 3 --> for pseudo 3D games, MEMORY 24000
;                               available command |3D
;
; ASSEMBLING_OPTION = 4 --> future use

let ASSEMBLING_OPTION = 0
;-----CODIGO-----
;includes the 8bp library and the music playerWYZ
read "make_codigo_mygame.asm".

;-----MUSICA-----
read "make_musica_mygame.asm";
includes the songs.

; ----- GRAPHICS -----
This part includes images and animation sequences.
; and the sprite table initialised with those images and sequences read
"make_graficos_mygame.asm".

```

Each of these three files is in charge of assembling different things and for example the graphics file invokes other files such as the image file, the sequence file, the route file and the world map.

Always use the assembly option that gives you the most free memory. If your game is a maze game, use option 1, and if it is a scrolling game, use option 2. If it is a pseudo3D game, use option 3. In general I don't recommend you to use option 0 because it is the one that gives you the least free memory and you will probably be able to use one of the other options.

## 6.2 Structure of the image file

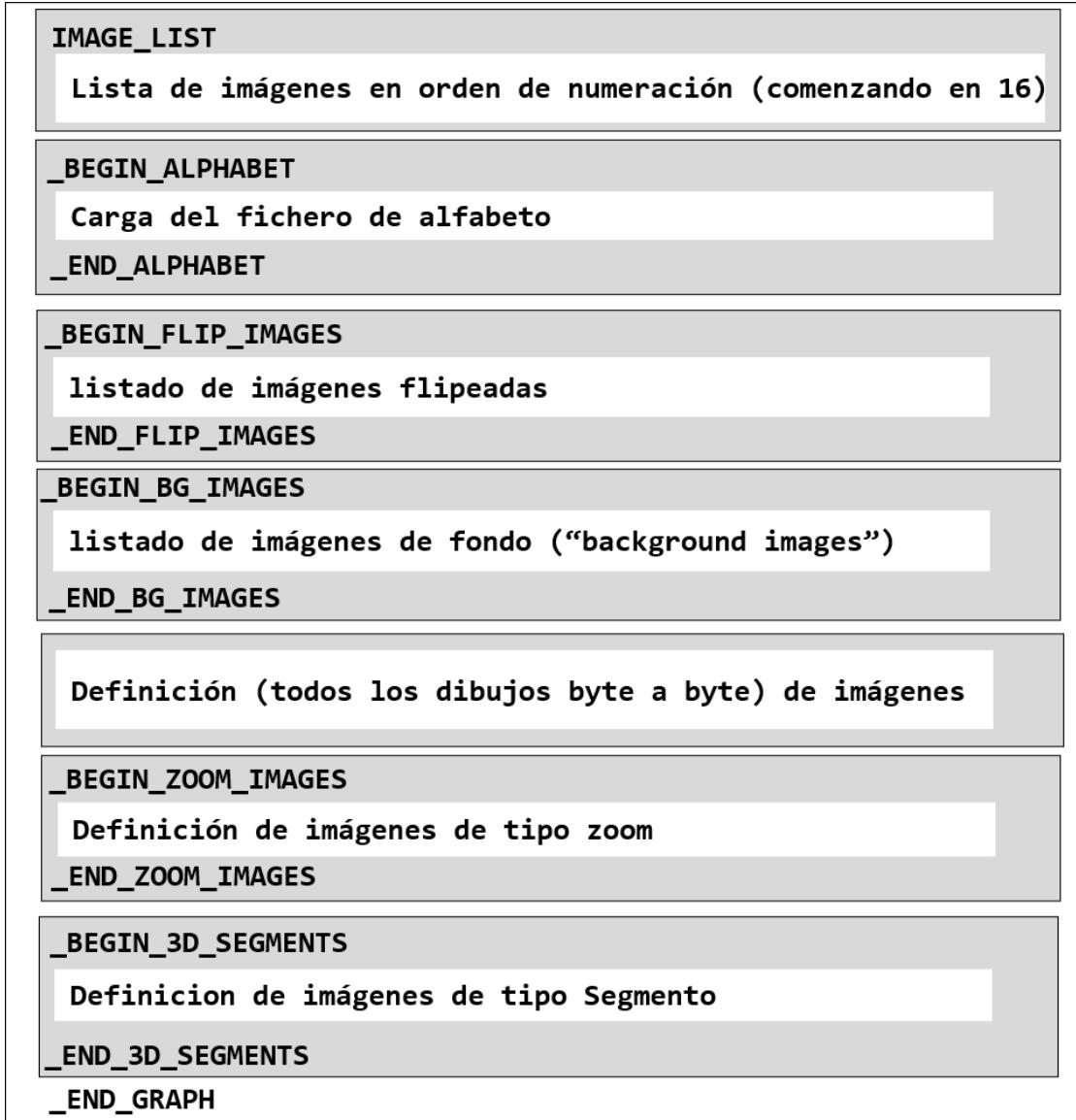


Fig. 15 structure of the image file

## 6.3 Animation sequence file structure

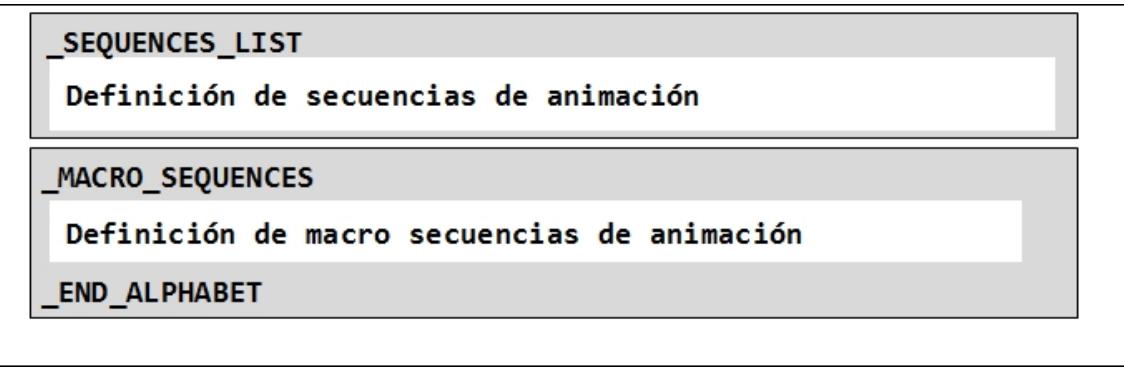


Fig. 16 Animation sequence file structure

## 6.4 Structure of the routing file

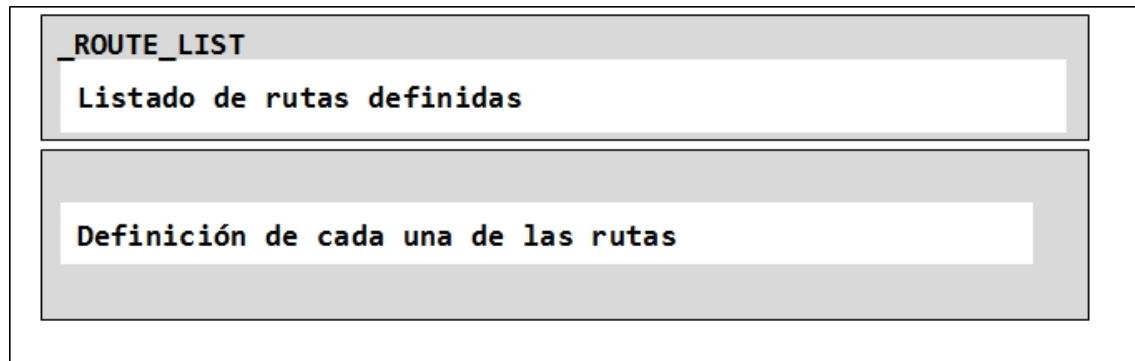


Fig. 17 Structure of the routing file

## 6.5 Structure of the world map file

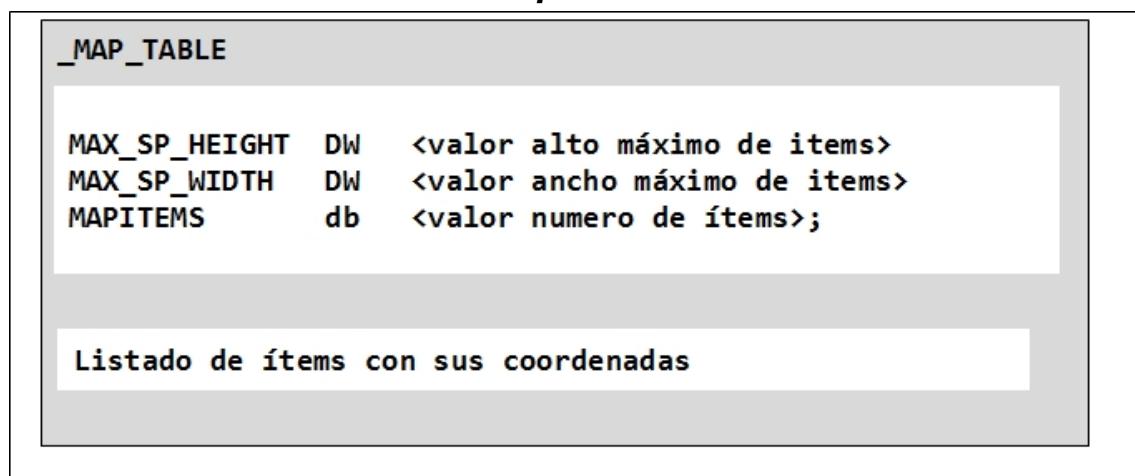
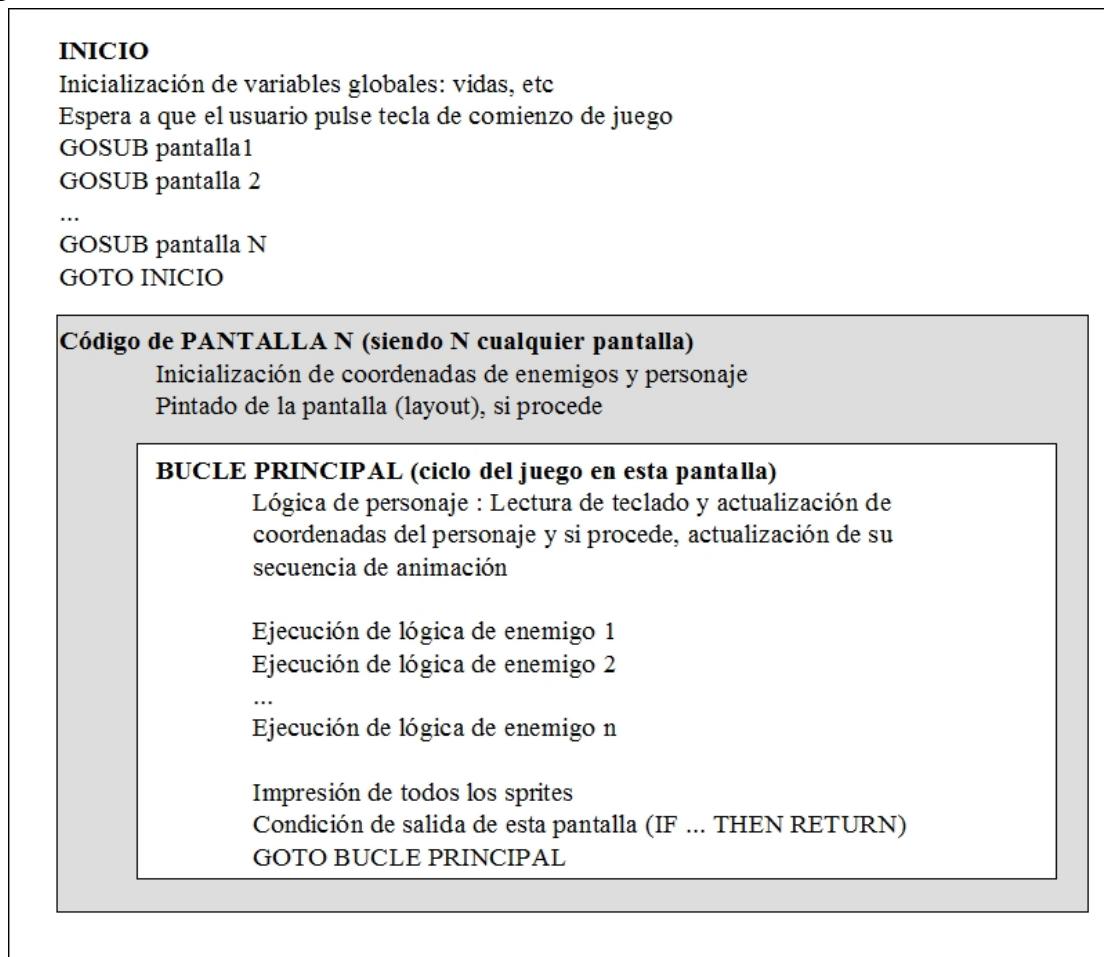


Fig. 18 World map file structure

## 7 Cycle of play

An arcade, platform, adventure video game generally has a similar type of structure, in which certain operations will be repeated cyclically in what we will call a "game cycle".

In each game cycle we will update sprite positions and print the sprites on screen, so that the number of game cycles that are executed per second equals the "frames per second" (FPS) of the game. The following pseudo-code outlines the basic structure of a game



*Fig. 19 Basic structure of a game*

If the enemy logic is too heavy due to too many enemies or too complex, this will consume more time in each game cycle and therefore the number of cycles per second will be reduced. Try not to go below 10fps for the game to maintain an acceptable level of action.

### 7.1 How to measure the FPS of your game cycle

To know if your game has an acceptable level of action, nothing better than playing it, and if you like it, then it will be fine. However, you may want to measure exactly how many frames per second your game is capable of generating, because then you can make decisions in the logic of your program and measure how much those programming decisions hurt or benefit it.

What we will do to measure is simply take note of the instant of time before starting the first game cycle, at the beginning of the "N-screen code". Then we will take note of the time after a few game cycles and do a simple division. Let's look at it step by step:

**A=TIME : rem this line stores in the variable A the time in 1/300 fractions of a second.**

The number to be stored in A can be a very large number, in fact, it can be larger than what an integer variable such as "A" is capable of storing. So that the assignment does not produce an error, it is convenient to reset the AMSTRAD's timer, which is started every time the machine is started. To reset it, before assigning the variable "A", simply run:

**On a CPC 6128**

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

**On a CPC 464**

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

To differentiate which machine your programme is on, you must deactivate the music and consult an address with PEEK.

**|MUSIC: If peek(&39)=57 then Model=464 else model=6128 If  
model=464 then ...**

With this you will have set to zero the memory addresses where the AMSTRAD stores the timer. Then, we run as many game cycles as we want, and we control which cycle we are in with the variable "cycle", which we will increase by one unit in each cycle. After exiting that phase or screen, we execute :

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

And now we have the FPS of our game. I'll put it all in order for you below:

**Rem assume we are in a 6128**

**POKE &b8b4,0: POKE &b8b5,0: POKE &b8b6,0: POKE &b8b7,0 A=TIME**

**<here go to the programme that runs your  
cycle of cycle, including cycle=cycle+1 >**

**We will get here after the output condition of the display**

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

**PRINT "FPS =";FPS**

To make it clear: The time elapsed from the start of the programme until it has finished is TIME-A expressed in 1/300 fractions of a second. To convert it to seconds, divide by 300.

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

If n cycles have been executed in these seconds (for example), then one cycle has taken:

**Tc= 300\*(TIME-A)/n**

And the number of cycles that can be executed in one second (the FPS) is the inverse, i.e.  
**FPS = 1/Tc = n\*300/(TIME-A).**

## 8 Sprites

### 8.1 Editing sprites with SPEDIT and assembling them

Spedit (Simple Sprite Editor) is a tool that will allow you to create your own character and enemy images and use them in your BASIC programs.

Spedit is made in BASIC, and it is very simple, so you can modify it to do things that are not contemplated and you are interested in. It runs on the Amstrad CPC, although it is designed to be used from the winape emulator.

The first thing to do is to configure winape to output the printer output to a file. In this example I have put the printer output to the file printer5.txt

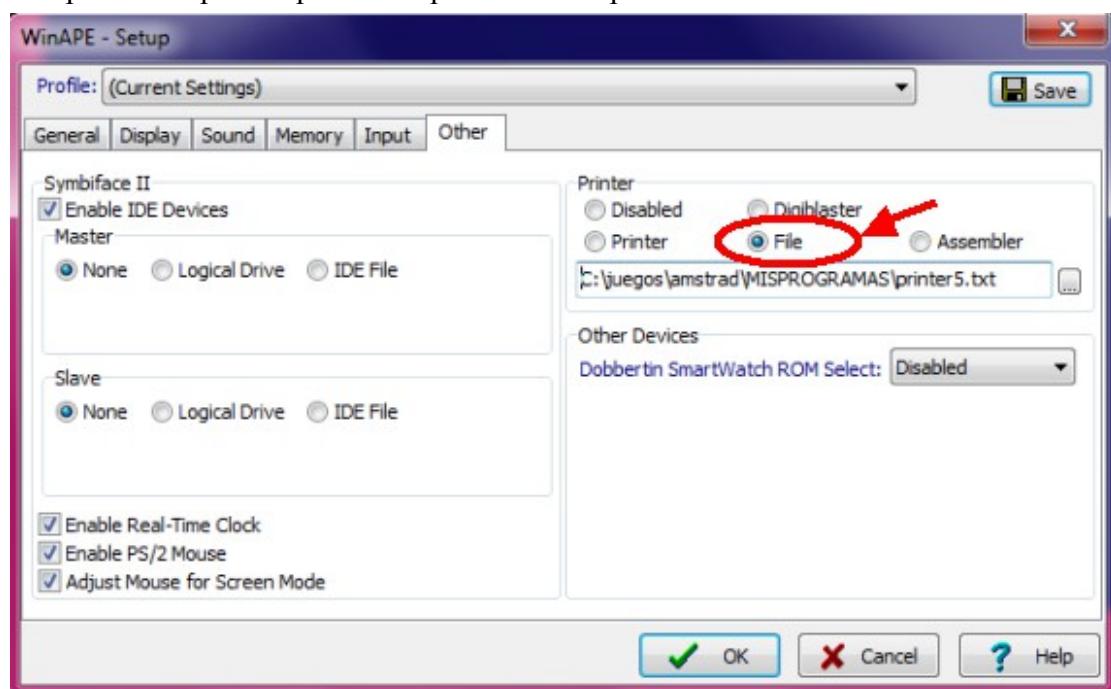


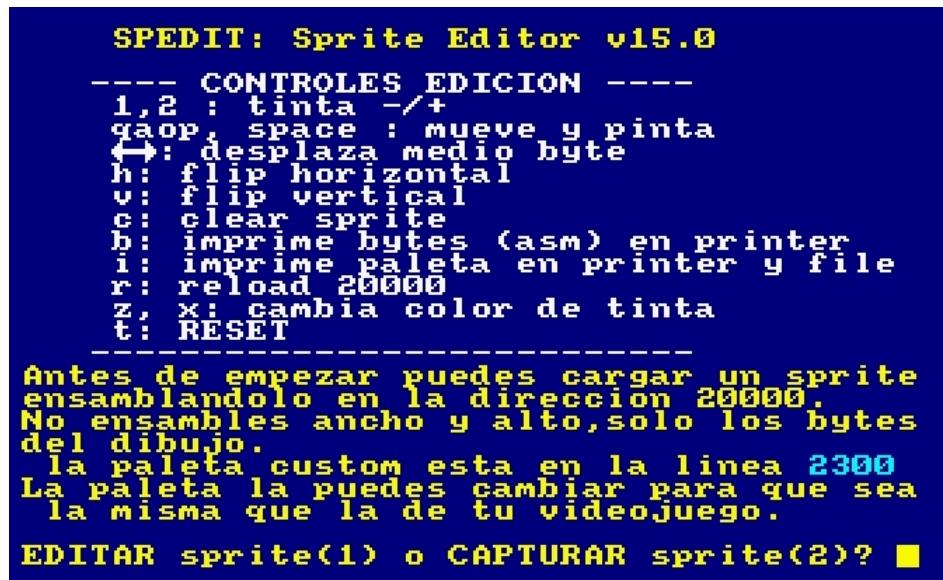
Fig. 20 Redirecting the CPC printer to a file with Winape

When you run SPEDIT you will get the following menu, where you can choose between editing a Sprite or capturing a Sprite from an image file (.scr).

Sprite capture is available from SPEDIT V14. In case you want to capture a sprite, you need to have an image file (.scr) on disk, which is just a binary file with 16384 bytes. It can be an image from a game that you have captured where there are images of characters that you like and you don't want to spend time editing.

In case you choose to edit a Sprite, the program asks you for the palette to use. You can choose a default palette or one of your own that you want to define. You can also use one you have previously saved (it is always saved in pal.dat, just press "i" while editing a sprite). If you decide to define your own palette, you will have to reprogram the BASIC lines where the alternative (or "custom") palette is defined.

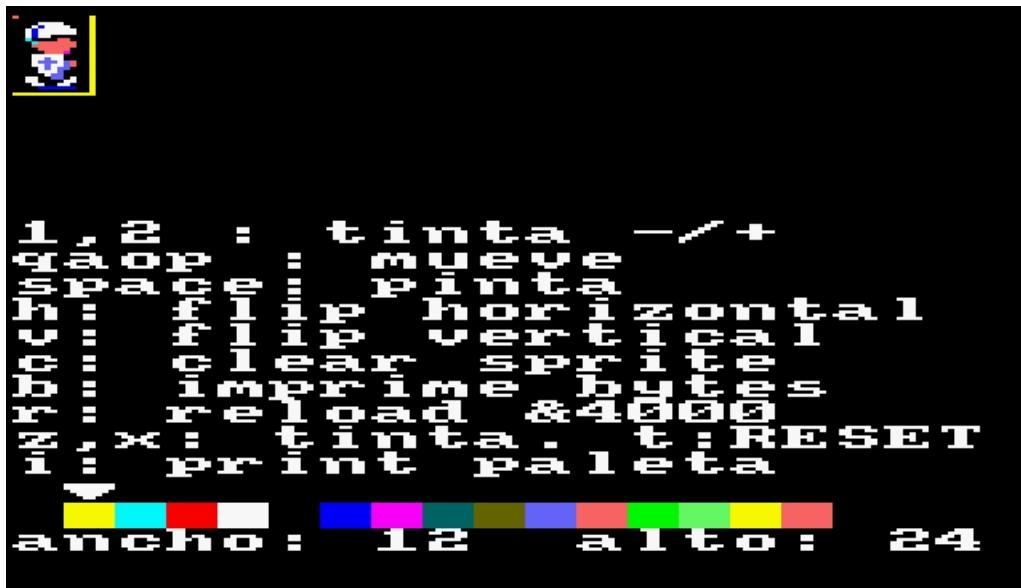
is a subroutine that is invoked by GOSUB when you press "2" in the answer to the question about which palette you want to use.



*Fig. 21 SPEDIT initial screen*

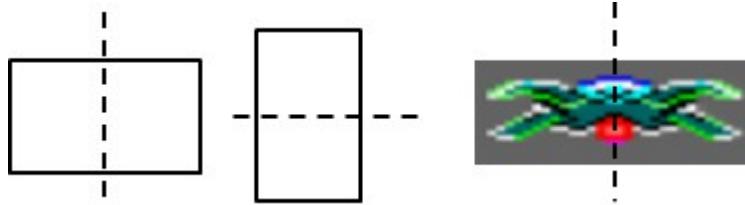
The tool allows you to choose mode 1 or mode 0. Once in edit mode it allows you to edit drawings, with help on the screen. You handle a flashing pixel and the coordinates where you are and the value of the byte you are in are shown at the bottom.

When it asks for the width and height of the sprite, remember that **the maximum height of a sprite in 8BP is 127 lines**, as well as its maximum width in bytes. Also note that the width is asked in pixels, but you should know that the editor internally works in bytes, so, if you are going to make an image in mode 0, the width must be an even number (one byte = 2 pixels) and if you are going to make an image in mode 1, the width must be a multiple of 4 (one byte = 4 pixels).



*Fig. 22 SPEDIT edit screen*

SPEDIT allows you to "mirror" your image to make the same figure walking to the left effortlessly, just by pressing H (horizontal flip) and the same can be done vertically. It also allows you to "mirror the image" with respect to an imaginary axis located in the centre of the character, both vertically and horizontally. This is very useful for symmetrical or nearly symmetrical characters, where a drawing aid is always useful.



*Fig. 23 symmetrical sprites with SPEDIT*

Since version 11 of SPEDIT, AMSTRAD mode 1 is supported, so you can edit sprites in mode 1 without any problem, and use the mirroring mechanism as well.



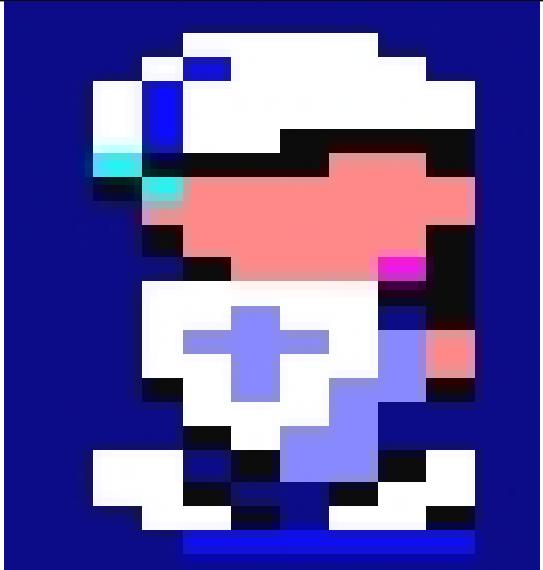
*Fig. 24 editing sprites in MODE 1 with SPEDIT*

Once you have defined your dummy, to extract the assembly code you must press the "b". This will send to the printer (to the file we have defined as output) a text like the following, to which you can add a name, I have called it "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 -----

```



Notice how I have always left one byte to the left at zero. I have done this so that, when the soldier moves to the right, it "erases itself", otherwise it would leave a trace, "smudging" the screen as it moves forward.

*Fig. 25 Soldier in .asm format*

Once you have made the first frame of your soldier, you can leave the work and continue another day. To start from the soldier you have drawn and continue retouching it or modify it to build another frame, you can assemble the soldier at address **20000**, removing the width and height. Once assembled from winape, you tell SPEDIT that you are going to edit a sprite of the same size and once you are in the edit screen press "r" (reload). The sprite will be loaded from address **20000**, which is where you have "assembled" it.

A big part of a game's appeal is its sprites. Don't skimp on this, do it slowly and tastefully and your game will look much better.

```

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 -----

```

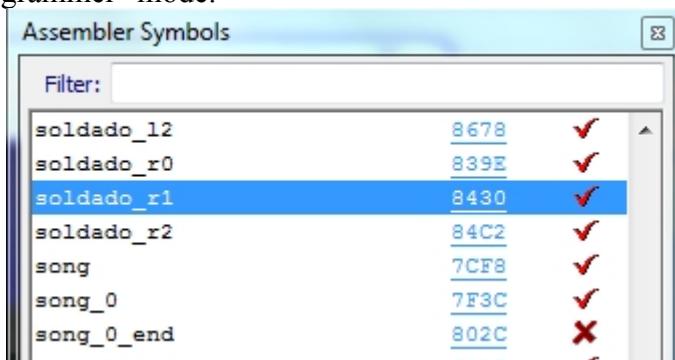
With this you know what it means to "assemble" a sprite. It is simply to put the bytes of data that constitute it in consecutive memory addresses, in this case starting with **20000**.

SPEDIT occupies very little memory and that address is far away from the program so there is no problem that by assembling it we are "damaging" the SPEDIT program.

*Fig. 26 Graphics assembly*

To know in which memory address each image has been assembled, use from the winape menu: Assemble->symbols

With this you will see a relation of the tags you have defined, like "SOLDADO\_R1" and the memory address (in hexadecimal) from which it has been assembled. To transform the addresses from hexadecimal to decimal you can use the windows calculator in "programmer" mode.



*Fig. 27 detail of what symbols shows in Winape*

Once you have made the different animation phases of your soldier, you can group them into an animation "sequence". Animation sequences are lists of images and are not defined with SPEDIT. With SPEDIT you simply edit the

"frames". In a later section I will explain how to tell the 8BP library which set of images constitute an animation sequence.

The images you make for your game are saved in a single file, called "images\_mygame.asm", for example. This file starts with a list of images that you can reference in the 8BP commands in BASIC with an index, regardless of the address where they are assembled, for example:

```
IMAGE_LIST
The first image will always be assigned to index 16, the next image to
index 16 and the following image to index 16.
17 and so on
;-----
dw SOLDIER_R0; 16
dw SOLDIER_R1; 17
dw SOLDIER_R2; 18
```

Once all the images are done you can assemble the library together with the music and graphics.

**VERY IMPORTANT:** make sure not to exceed 8440 bytes of graphics. To do this, check where the "\_END\_GRAPH" tag is assembled, which must be less than 42040 (since  $42040 - 33600 = 8440$  bytes). If it is assembled at a higher address then you are "crunching" addresses needed by the BASIC interpreter and the computer may crash. In case you need more memory for graphics you should assemble the "extra" graphics in an unoccupied memory area, e.g. 22000, and use a MEMORY 21999 in your program, reducing the memory available for BASIC.

## 8.2 Print a sprite

Let's look at the basics of printing a sprite. Let's assume you have drawn a soldier and included it in the file images\_mygame.asm. Let's assume it's your first image and therefore has the identifier 16.

In 8BP you have 32 sprites (numbered from 0 to 31). You can assign an image to any sprite with the |SETUPSP command. This command allows you to modify many attributes of a sprite, not just its image. The attribute you want to modify of the sprite is the second parameter of the SETUPSP command.

**|SETUPSP, <sprite id>, <parameter> , <value>, <sprite id>, <parameter> , <value>, <value>.**

To assign an image you must use parameter 9. A very simple program that prints a sprite would be the following:

```
10 MEMORY 23499
20 CALL &6B78: REM installs RSX commands
30 DEFINT A-Z : REM integer numerical variables (faster)
40 |SETUPSP,31,9,16: REM assigns image 16 to sprite 31
50 x=40:y=100: REM coordinates where we want to print
60 |LOCATESP,31,y,x: REM places sprite 31
70 |PRINTSP,31: REM prints sprite 31
```

This would be the result



*Fig. 28 Printing a sprite on screen*

You will use the **SETUPSP** command a lot and you will gradually get to know it in depth. The SETUPSP parameters are not just any number. There are 7 possible values and they are the following (0,5,6,7,8,9,15):

- Parameter 0: change the status byte of the sprite
- Parameter 5: change Vy. Vx can also be changed at the same time if we add it at the end as an extra parameter ( like this : SETUPSP, <id>,5, Vy,Vx )
- Parameter 6: changes Vx
- Parameter 7: change sequence (takes values 0..31)
- Parameter 8: change frame\_id (takes values 0..7)
- Parameter 9: change image. The specified image can be one of the initial list of images in the images\_mygame.asm file,
- Parameter 15: change path (occupies 1bytes)

As you read through this manual you will come to understand the meaning of the attributes of a sprite, and how to use them according to your needs.

### **8.3 Sprite flipping**

On many occasions you will need to draw characters walking in different directions, with different images for each case. The image of the sprite in the left direction will be the mirror image of the right direction. You can define two images and store them in memory, but since V33, there is a way to avoid RAM consumption for these images. This is called "flipped" images.



*Fig. 29 example of flipped images*

A "flipped" image is a mirror image of another image that has been created and included in the image file. By defining an image in this way, you avoid having to store it. To do this, you simply include a list of flipped images in the image file (which I usually call "images\_mygame.asm"). You will find at the beginning of the file a section delimited by the tags "BEGIN\_FLIP\_IMAGES" and "END\_FLIP\_IMAGES" for this purpose.

```
;_BEGIN_FLIP_IMAGES
here put images that are defined as other existing images but flipped
horizontally.
JOE_LEFT dw JOE_RIGHT; joe_left will be the flip-flopped version of
joe_right

I define the frames of the soldier on the left as flipped SOLDIER_L0 dw
SOLDIER_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
;
```

Flipped images can be used in the same way as normal images. Below you will find out how to create animation sequences, which you can build with both flipped and non-flipped images. To all intents and purposes, it is as if a "flipped" image is real, although it is a "virtual" image, which is not stored and is calculated as a mirror image of an existing image when printed. Flipped images are supported in both mode 0 and mode 1.

The disadvantage of the flipped images is that they are more expensive to print, specifically they take 1.8 times as long as a normal print, which could translate into a slower speed for your game. If your game is an arcade game (a shoot'em up) where you need maximum speed, my recommendation is not to use massively flickered images. However, in adventure games, screen passing games, maze games, etc. it is an excellent choice. In any case, try using them in your arcade, because if there are not many flips at the same time, the resulting speed can be very acceptable.

I have done horizontal flipping and not vertical flipping because normally a character walking to the left is the mirror image of the same character walking to the right, while walking up shows the back and walking down shows the chest and face. Therefore, vertical flipping is not as useful as horizontal flipping, and in the interest of reducing the size of 8BP, I have not included it among its capabilities.

**IMPORTANT:** flipping is not applicable to segment type images that can be used in 8BP pseudo-3D mode.

## 8.4 Sprites with overwriting

Since version v22 of 8BP it is possible to edit transparent sprites, i.e. sprites that can fly over a background and reset it when passing. To do this, sprites that enjoy this possibility must be configured with a "1" in the overwrite flag of the status byte (bit 6). In the next section, the status byte will be explained in detail. Let's see how to edit a sprite with this capability with SPEDIT.

Many games use a technique called "double buffering" to be able to restore the background when a sprite moves across the screen. This is based on having a copy of the screen (or game area) in another area of memory, so that even if our sprites destroy the background, we can always look in that area to see what was underneath and restore it. Actually, that's the basic principle, but it's a bit more complex. It is printed in the double buffer (also called "backbuffer") and when it is all printed, it is either dumped to the screen or the starting address of the video memory is switched from the original screen address to the new one, the double buffer address. The switchover is instantaneous (depending on the type of machine). To build the next frame, the original screen address is used where the video memory is no longer pointing. There the new frame is constructed and switched back, alternately, at each frame. These techniques, although they work very well, have a couple of disadvantages for our purposes: they take more CPU time and consume much more memory (up to 16KB extra), leaving us very little memory for our BASIC program. If a game is developed entirely in assembler, this is not such a big deal because 10KB of assembler goes a long way, but 10KB of BASIC is too little. Some videogames reduce the game area in order not to use so much memory, but that makes them a bit poorer.

The solution adopted at 8BP is inspired by programmer Paul Shirley (author of "Mission Genocide"), but is slightly different. I will tell the story of 8BP directly:



Fig. 30 Sprites with overwriting in 8BP

The idea is that **the background is never destroyed by the sprites passing over it**, so there is no need to save it. This apparent "magic" has its logic: it consists of "hiding" the background colour in the colour of the sprite that is painted over it.

On the AMSTRAD a pixel of mode 0 is represented with 4 bits, so up to 16 different colours are possible out of a palette of 27 colours. So, if we use one bit for the background colour and 3 bits for the sprite colours, we will have a total of 2 background colours.

7 colours + 7 colours + 1 colour to indicate transparency = 9 colours in total. This will allow us to "hide" the background colour in the colour of the sprite, although we pay the price of reducing the number of colours from 16 to only 9. However, certain ornamental elements on the game screen can have more colour, as the sprites will not pass over them (such as the leaves on the trees or the roof in the example below), so we can get a certain amount of colour in our game.

To edit this kind of sprites we must use a proper palette, of 9 colours, where for each sprite colour two binary codes are used (corresponding to the 0 and 1 of the background bit). In SPEDIT if you choose the palette option "2", you will have a palette defined that way, although you can change it to your liking. It is constructed like this:

```

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 Example overwrite palette*

As you can see, after colour 0 and 1, all colours are repeated twice. You can build your own palette in this way. You can help yourself by consulting the appendix of this manual dedicated to the colour palette.

	=  Color de fondo	Paleta ejemplo
	=  Color de sprite	0000 = 0001 = 1100 = 1101 =
<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>		

The technique could be summarised by saying that **the background is never actually destroyed by the sprites**, but is "hidden" in the sprites themselves that are printed on the background.

*Fig. 32 Overwrite mechanism in 8BP*

With the SPEDIT editor you can modify the palette to your liking without the need to edit it manually with INK commands, and you can export it to copy it to our BASIC programs. The export is done by sending the INK commands that make up the palette to the printer (the printer is redirected to a file from winape). We have the z/x keys to alter the palette and the "i" option to export it to the output file. This is an example of what it exports (it is a palette without overwriting):

```
' ----- BEGIN PALETTE -----  
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 -----
```

Pressing the "i" key also saves the "pal.dat" file to disk (the .dsk) so that you can load it later by choosing option 3 to answer the palette choice question.

The sprites you use to build the background drawings can only have the colours 0 and 1, but the sprites you use for ornamentation, where the moving sprites are not going to pass through, can use all 9 colours.

You can also increase the colourfulness of the scenery with sprite elements instead of backgrounds, such as the green cauldron in the example above. This way you can get very colourful results.

Ink 0001 has a "special" use. If you edit a sprite that does not use the overwrite flag, ink 1 will simply be a colour. But if you edit a sprite with an active overwrite flag in its status byte, when printing, those pixels will be left unpainted, respecting whatever is underneath. This allows collisions between sprites not to be "rectangular", but to preserve the shape of the sprite.

code	Meaning
0000	Background colour 1. If a sprite uses it and you activate the overwrite flag, it means "transparency", i.e. printing by resetting the fund
0001	Background colour 2. If a sprite uses it and you turn on the overwrite flag, it no longer means a colour, it means "do not print". Whatever is in that pixel is respected, e.g. a pixel coloured by another sprite previously printed with the that we are overlapping.
0010	sprite colour 1
0011	
0100	
0101	sprite colour 2

code	meaning
0110	sprite colour 3
0111	
1000	colour 4 of sprite
1001	
1010	sprite colour 5
1011	
1100	sprite colour 6
1101	
1110	sprite colour 7
1111	

9 colours in total:

- 2 in the background
- 7 for sprites (actually 8 but one - 000- means transparency)
- The elements ornamental elements can use all 9.

Next, I'm going to show you a sprite where I have painted with 0001 ink what is not going to be painted, that is, where the background is not even going to be restored, because with the rest of the pixels at 0000 it is enough to erase the trace of the sprite while it moves.

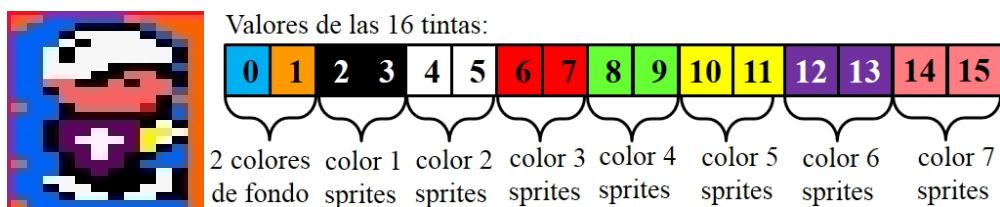


Fig. 33 sprite and palette designed for overwriting

#### IMPORTANT, WHEN EDITING YOUR SPRITES WITH OVERWRITE:

Do not use the background bits for your sprite inks, unless you are looking for special effects as described later in this chapter. If you don't follow this rule, you may notice "weird" effects. That is to say:

- If the background is 1 bit, then colour your sprites with inks ending in 0 (i.e. inks 2,4,6,8,10,12,14).
- If the background has 2 bits, then colour your sprites with inks ending in 00 (i.e. inks 0100,1000,1100 which are inks 4, 8 and 12 respectively).

As you can imagine, in the case of the cauldron, being a sprite that does not move and therefore does not erase itself, its entire outline is painted with the 0001 ink. This allows for perfect collisions, without rectangular shapes that make it evident that the sprites are

actually

are rectangles. The end result is as shown below in a multiple collision.



*Fig. 34 Multiple collision, ink effect 0001*

As you might have guessed, the collision is not only perfect, but also shows that the sprites have been ordered according to their Y-coordinate, so that the last one to be printed is the one located at the lowest position. This is done with a simple parameter when printing the sprites with the |PRINTSPALL command, which we will see later.

Printing operations with this mechanism are very fast, without the need to define what are known as "sprite masks". Sprite masks are sprite-sized bitmaps that serve to speed up printing operations. In this case they are not necessary. The following figure represents a typical mask associated with a sprite. First the AND operation is usually done between the background and the mask and then the OR is done with the sprite. In 8BP it is faster, because the sprite does not touch the bit destined for the background, so the OR operation between the background and the sprite respects the background while painting the sprite. If you don't understand this very well, don't worry, it is not important to understand it as it is not necessary in 8BP.

sprite	mask			
0	2	2	0	
2	3	3	2	
0	2	2	0	
0	2	2	0	
0	0	0	0	

<b>Metodo convencional:</b>				
Se imprime				
Fondo AND mask OR sprite				

<b>Metodo 8BP:</b>				
Imprimimos				
Fondo OR sprite				

*Fig. 35 In 8BP no masks are necessary.*

Printing sprites with the overwrite flag active is more expensive than printing without overwrite. Although it requires no mask and is very fast, it takes about 1.6 times as long as printing a sprite without overwrite. For that reason, use it when necessary, and don't use it if your game is not going to have a background drawing that the sprites must respect. The combination of overwrite and flipping is even more expensive (it consumes 2.2 times the time of a normal print without overwrite and flipping) so take this into account in your games.

#### 8.4.1 Using overwriting to improve sprite overlaps

A very valuable feature of the overwrite is that it allows the overlaps between sprites to be "perfect", because if when you edit the sprite (which you are going to overwrite) you use ink 1 around it, when you print it all the pixels that have ink 1 will be "overwritten".

will become "transparent", i.e. if you have previously painted another sprite, its pixels will be respected, resulting in "perfect" overlaps. You may be interested in this feature of overwrite sprites, even if your game does not require overwriting because it has a black or single-colour background.

Only those pixels that you have drawn with zero ink will be erased (by resetting the background). In the ball example, the zero pixels are the back pixels so you can erase it when you move it to the right.

With the following illustrative example you can understand it perfectly. You lose colour because there are only 9 colours, but the overlaps between sprites are very good. Be careful, because you also lose some printing speed.

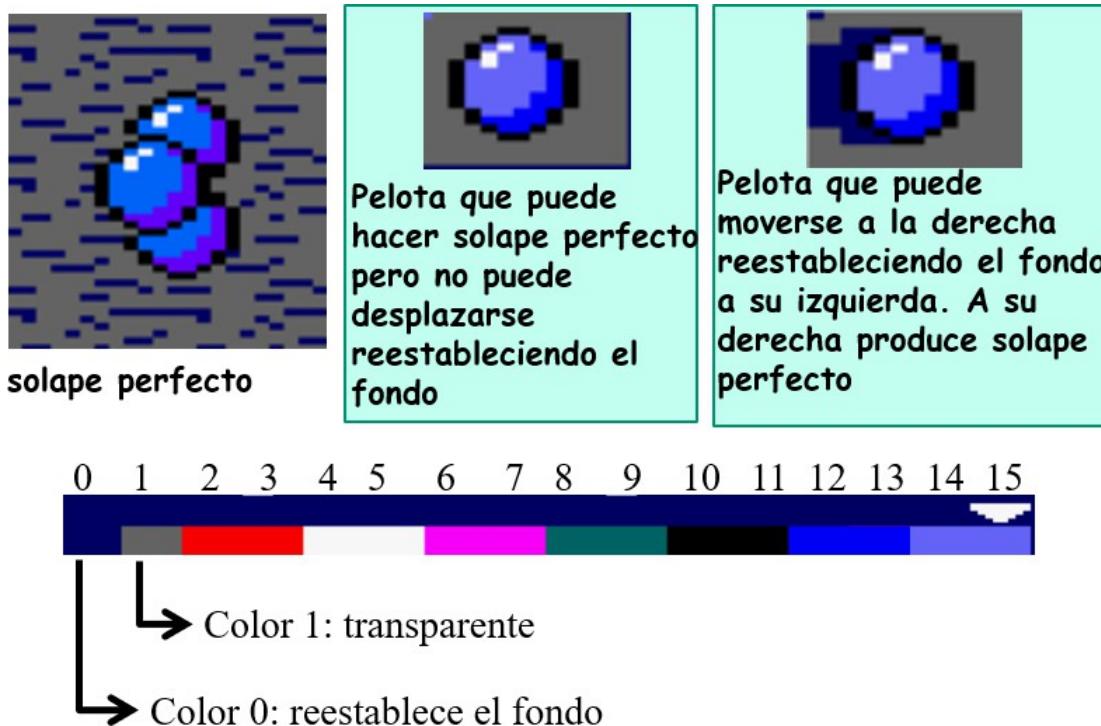


Fig. 36 Perfect overlaps

#### 8.4.2 Overwriting with 4 background colours

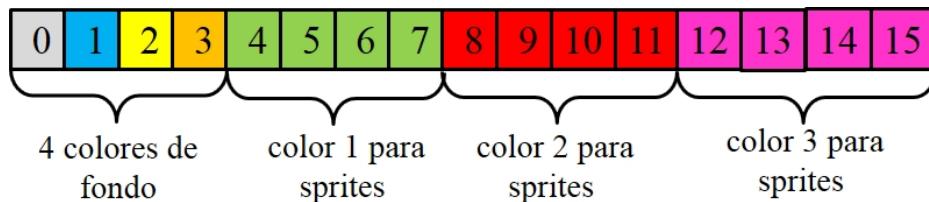
Since version V33 it is possible to choose the number of bits used for the background by invoking the |PRINTSP command specifying Sprite 32, which does not exist. You can choose 1 or 2 bits for the background, allowing 2 and 4 background colours respectively.

**|PRINTSP, 32, <background bit number>.**

Examples:

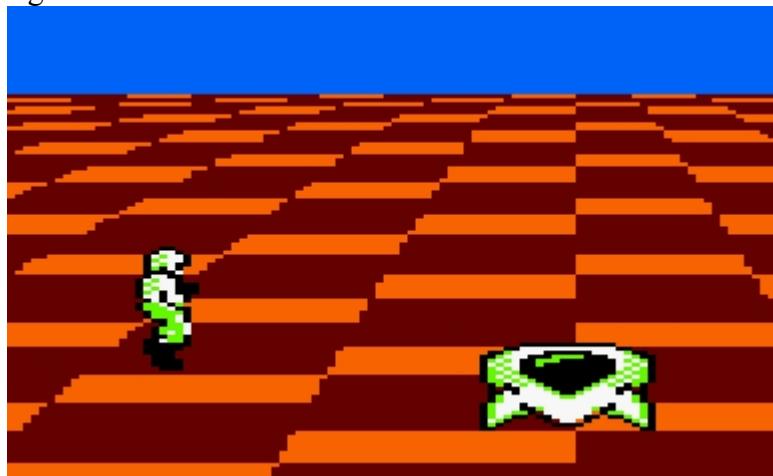
<b> PRINTSP, 32, 1 : ' with 1 bit background we have 2 colours for the background</b>
<b> PRINTSP, 32, 2 : ' with 2 bit background we have 4 colours for the background</b>

Once this command is invoked, the 8BP library is configured to take into account the number of bits to be used as background bits. If we set 2 background bits, our colour palette will have to be consistent with this circumstance. An example is shown below



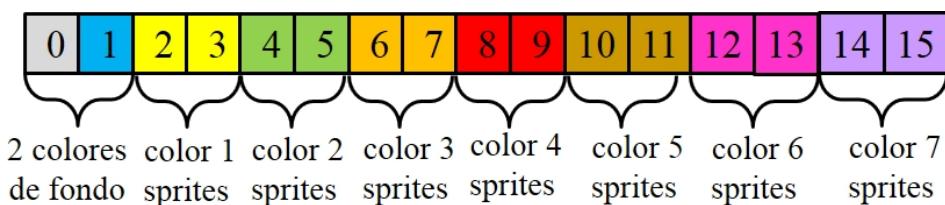
*Fig. 37 Example of a palette with four background colours (2 bit background)*

In this example you have 4 background colours and three sprite colours. Just like when using 1 bit for the background, when defining your sprites you should keep in mind that 0 ink means transparency and 1 means not resetting the background, allowing for non-rectangular sprite shapes. In the following example the 3 colours chosen for the sprites are black, light green and white.



*Fig. 38 example of a palette game with up to four background colours (2 bit)*

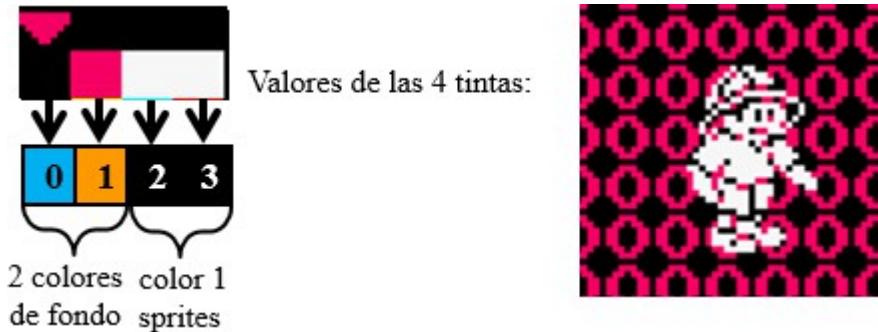
Although with two bits for the background you can get nicer decorations, the disadvantage is that you only have 3 colours left for the sprites, while if you use 1 bit for the background, you have up to 7 colours for your sprites.



*Fig. 39 Example of a palette with two background colours (1 bit background)*

### 8.4.3 Overwrite in MODE 1

Since version V34 of 8BP, it is possible to use sprites with overwrite in MODE 1. Here we have a very strong limitation because, although we have two colours for the background, we only have one colour for the sprites.

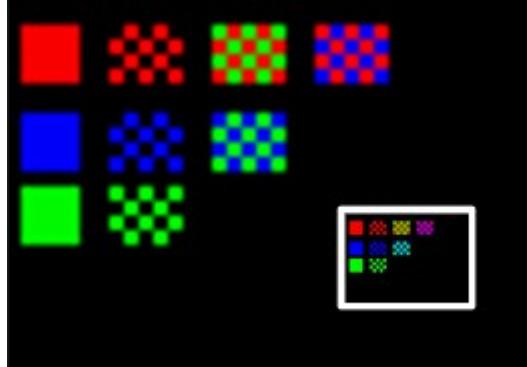


*Fig. 40 Example of mode 1 palette*

It is easy to make the mistake of thinking that the sprites have any colour and, moreover, black. This is not the case. Black is another colour and as such it must consume two inks with this mechanism. We only have two inks for the sprite and we have used them on the white (in this example). As you can see, the character where it is not white is transparent and not black.

Despite this strict limitation, if you work hard you can make some very attractive sprites in MODE 1, and if you change the background colours on each screen, and use colour mixes (lattices) on the game markers, you can achieve a very satisfactory result.

By using blending in MODE 1 you can simulate 10 colours with only 4 colours. The lattice colours are merged and for example, green + red looks yellow. You may not see it as convincing in this picture, but on your Amstrad screen you will see it well.



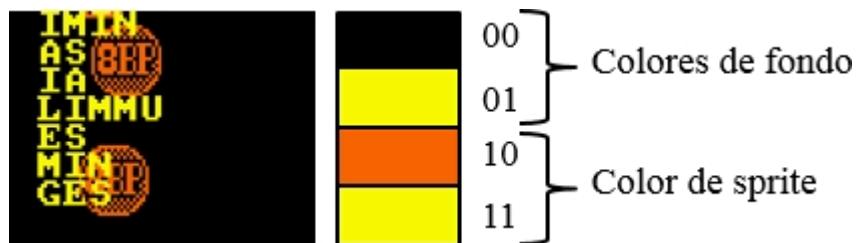
#### 8.4.4 How to paint sprites "behind" the background

The same mechanism for printing sprites on top of the background can be used for painting behind the background.

As we have already seen, to print in front of the background we use bits that are not used in the background, so that, although we reduce the number of colours, we do not damage the background bit(s). If we use one bit for the background, we will have to use two inks to represent the same sprite colour: one with the background bit set to zero and one with the background bit set to 1.

However, if instead of assigning the same colour to these two inks, we assign the same colour as the background to the one with the background bit set to 1, then if the sprite overlaps with the background, the background colour will be seen, giving the impression that the sprite passes behind it. This works in both MODE 0 and MODE 1.

In the following example a bit is used for the background, which consists of yellow letters on a black background. The sprites are "coins" which, as you can see, have apparently been painted behind the background.



*Fig. 41 Sprites printed "behind" the letters*

#### 8.4.5 How to use more colours with overwriting

If you have done your first tests with overwriting and you need more colours in your game, there are three ways to achieve this, but you need to understand the 8BP method:

- 1) Use some sprites with overwrite and some without overwrite
- 2) Use sprites whose colour depends on the background (conditional colour).
- 3) Use a sprite as background

Let's look at these three "tricks" one by one:

##### Use some sprites with overwrite and some without:

This is the simplest and most commonly used of the three tricks. Suppose only one of your sprites requires overwriting and consumes 3 colours. That means you must allocate 6 inks to this sprite. However, if the rest of the sprites do not require overwrite, you can print them without overwrite and use more colours on them, ie:

2 inks for the background (2 colours)  
 6 inks for the Sprite with overwrites (3 colours) 8  
 inks for the other sprites (8 colours)

In this case, you can use a total of  $2+3+8 = 13$  colours!!!!. You simply have to activate the overwrite flag on the sprite that needs it and leave it inactive on the other sprites. On the sprites that don't use overwrite you can use all 13 colours, on the sprite with overwrite you would use 3 and on the background you would use 2.

Other examples are possible. For example, if sprites with overwriting need 4 colours, then they will use 8 inks. In addition we will have 2 inks for the background and the remaining 6 inks can identify 6 different colours, i.e. we can use a total of 12 colours.

##### Use sprites whose colour depends on the background (conditional colour):

It consists of using a palette where instead of repeating each colour, we use two different colours. In that case, when the background is zero we will have a sprite of one colour and when the background is 1 we will have another colour. This can be useful to draw white birds flying over a blue sky (colour 0) while red bears walk on a brown ground (colour 1). In other words, we can use more colour as long as the texture of the sky or the ground

The bear will not have too many colour changes, because every time the bear passes over a blue background pixel we will see a white pixel, and if a bird passes over a brown background pixel we will see a red pixel. Let's see an example:

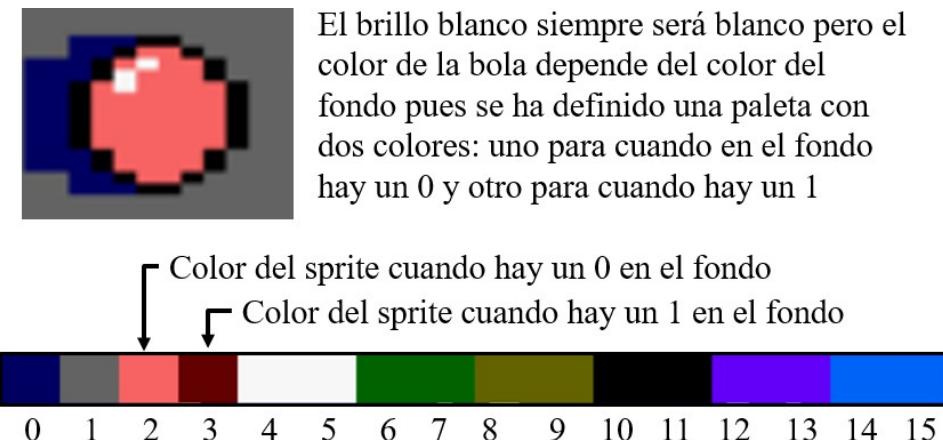


Fig. 42 Sprites created with a "conditional" colour

Once the sprite with conditional colour has been created, in the following image we can see the effect it has when printed in two areas of the screen, one with background 0 and the other with background 1.

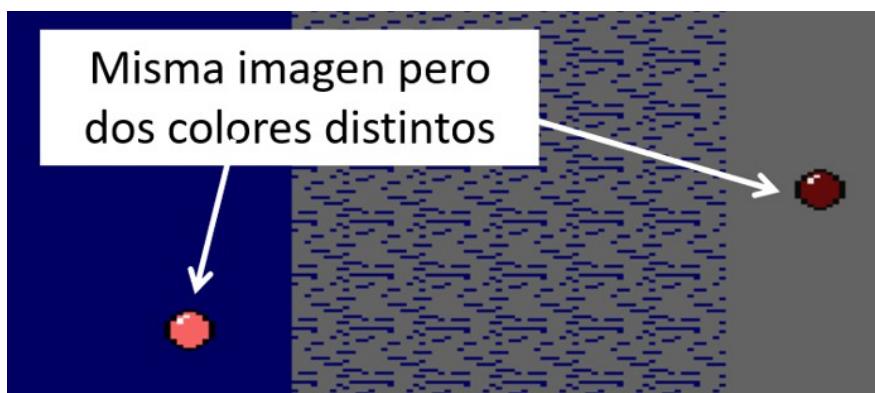


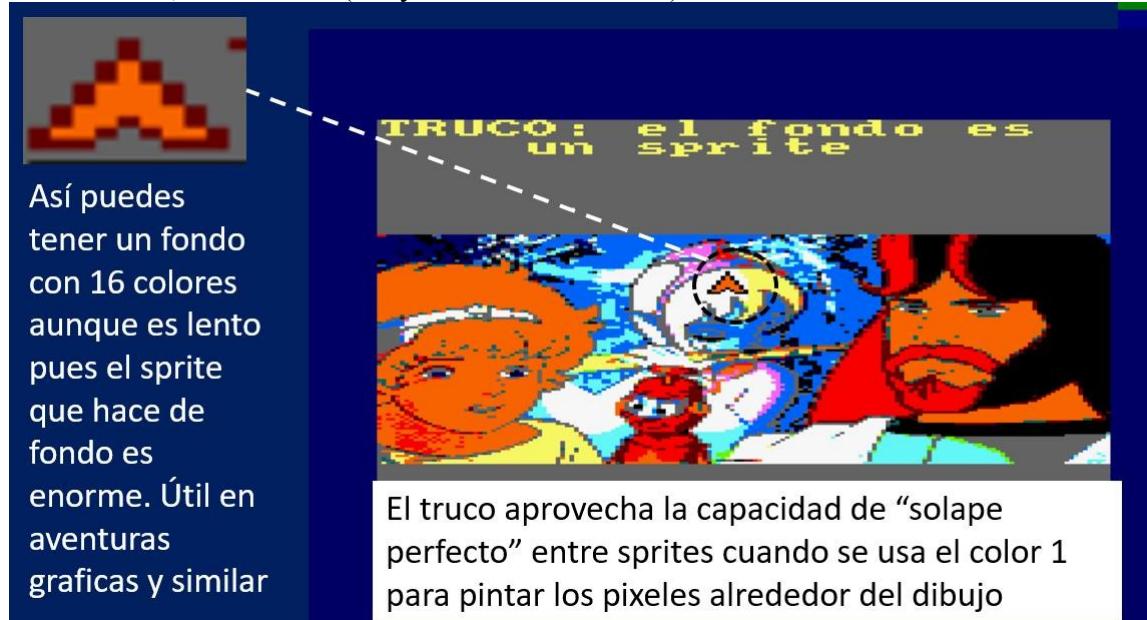
Fig. 43 "Conditional" colour effect

### Use a sprite as background:

This cheat allows you to use up to 16 background colours with overwrite. The trick is based on the fact that, when overwrite is activated on a sprite, all pixels defined with "1" will respect the value of the pixel on the screen, whether it is a pixel from a real background or from a previously printed sprite. We will print in each cycle both the sprite that is the background and the sprites that will be overprinted. Or at least in each cycle that we move them.

The sprite that we use as background, we will print it **without activating the overwrite**, while the sprites that we print on top of it will have it activated. The only drawback is that if the background sprite is too big, it will take a long time to print and the number of fps of the game will not allow to make an arcade game, although it can be useful for adventure games. Also, **remember that in 8BP the maximum height of a sprite is 127 lines**. The maximum width is not a problem as it is also 127 bytes and the screen is only 80 bytes wide.

We will create the sprite that we are going to print on top surrounded by ones, without zeros as it will not reset the background when moving. Each cycle will print both the background sprite and our little sprites on top of it. In this example we have drawn a kind of pointer or arrow that you control with the keyboard, on a background that occupies half a screen, that is 8KB (80bytes wide x 100 lines).



*Fig. 44 A sprite as background*

As the pointer sprite has overwriting, it will suffer the effects of the previous trick (conditional colour) unless you define some "double" colours not to suffer that effect. That is, if for example you repeat a colour so as not to suffer that effect on a sprite, then you will have a palette of 15 different colours, not 16. That is, somehow you have to apply the principle of the first trick: one sprite without overwrite as background and one or more sprites with overwrite that are printed on top of it. The more colour the background has, the less colour your overwrite sprites will have. For example (other combinations are possible) you can use:

- **Background:** 2 background colours + 8 colours + 3 repeated colours = 13 colours
- **Overprinted sprites:** 6 inks = 3 repeated colours

One way to speed up the speed of the "giant" background sprite is to break it into strips. For example, into 8 horizontal sprites of height 16. Horizontally stretched sprites print faster than vertically stretched sprites. In that case you only need to print the stripe or pair of strips that intersect your pointer sprite.

## 8.5 Sprites with background images

Background images are a feature of 8BP V42. The idea is that in a scroll, trees or houses can pass under your plane and not cause the plane to flicker. Even if the plane has transparent printing and is painted on the background, if the background moves, it will not respect the sprite and will cause flickering. With this new capability, you can make games with scrolling where your protagonist or ship passes over things and **there is no flickering**. To understand this better, you need to understand the scrolling mechanism of 8BP, which is explained below.

Included in the 8BP demo set is one that demonstrates the effect of using "background" images in your scroll.



Background images of houses

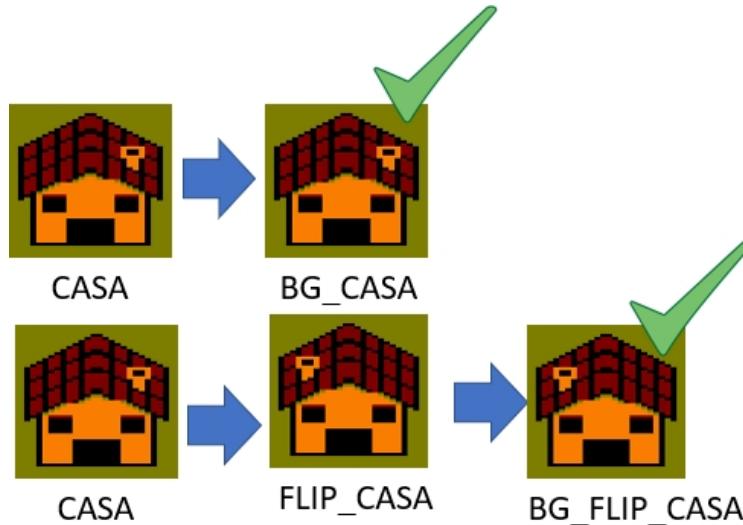
Without this capability, (before V42 version of 8BP) sprites could be overwritten without flickering as long as the background did not move, but the movement of the background (the scroll) caused unavoidable flickering. Since V42 it is possible to create background images and assign them to sprites.

What you need to do is simply assign the Sprite a background image, which in the image file must be contained within the tags:

```
_BEGIN_BG_IMAGES  
BG_HOME DW  
HOME  
BG_HOME_FLIP DW HOUSE_FLIP  
_END_BG_IMAGES
```

The **CASA** image is an image normally created with a particularity: it only uses the background colours. Thus, when a Sprite in a game has the **BG\_CASA** image assigned to it, it will automatically have a special type of transparency assigned to it, in which only the bits that represent the background colours will be modified and any Sprite above the drawing will be respected, avoiding flickering.

The process of creating background images is very simple, but also very strict: From an image (using only the background inks) you can build a background image. If you want to use a flipped image you must first flip the image and then build the background image with the flipped image.



In case you want to flip a background image, you must first flip it with the **FLIP\_IMAGES** section and then you can use it to build a background image. That must be the order and not the other way around. That is, you can't create an image, create a background image with it and then try to flip a background image.

The background images whenever they are assigned to a Sprite are printed with this special transparency and it doesn't matter if the Sprite has the transparency flag assigned or not in its status byte (we will now see what this is).

**IMPORTANT:** background images are expensive to print, and if you flip them, they are even more expensive. If your game has scrolling, try to use them for elements that your character or enemies will be flying over. For example, to speed up scrolling, use normal images on houses or rocks that appear on the sides of your scroll, which won't often be overlapped by sprites.

## 8.6 Sprite attribute table

The sprites are stored in a table containing a total of 32 sprites.

Each entry in the table contains all the attributes of the sprite and occupies 16 bytes for performance reasons, since 16 is a multiple of 2 and this allows accessing any sprite with a very inexpensive multiplication. The table is located at memory address 27000, so it can be accessed from BASIC with PEEK and POKE, but there are also RSX commands available to manipulate the data in this table, such as |SETUPSP or |LOCATESP.

Sprites have a set of parameters, the first of which is the status flags byte. In this byte, each bit represents a flag and each flag means one thing, namely whether the sprite is taken into consideration when executing certain functions. The following table summarises what happens if they are active (set to "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 in the status byte

To understand the power of these flags, let's look at some examples:

- **Bit 0:** print flag: our character or enemy ships will have it activated and in each game cycle we will invoke |PRINTSPALL and they will all be printed at the same time.
- **Bit 1:** collision flag: a fruit or a coin for example may not have a print flag, but may have a collision flag. This flag means that a "colliding" sprite can collide with the sprite that has this flag active.
- **Bit 2:** automatic animation flag: it is taken into account in |ANIMALL. In the case of the character, I recommend to deactivate it, because if I stay still, I don't have to change the frame.
- **Bit 3:** Automatic movement flag. Moves only when AUTOALL is invoked, taking into account its speed. Useful on meteors and guards that come and go.
- **Bit 4:** relative movement flag. All sprites with this flag move at the same time when invoking "|MOVEALL, dy, dx" very useful in formation ships and planet arrivals. It can also be used to simulate a scroll if you leave your character in the centre and pressing the controls scrolls houses or surrounding elements. It will look like your character is moving through a territory.
- **Bit 5:** collider flag. All sprites with this flag active are considered by the |COLSPALL function, when detecting their possible collision with the rest of the sprites.
- **Bit 6:** overwrite flag: if this flag is active, the sprite will be able to move over a background, resetting it on passing. This is an advanced option and involves the use of a specially prepared colour palette, more limited in number of colours. Overwriting comes at this "price".
- **Bit 7:** path flag: if this flag is set, the |ROUTEALL command will allow you to move a sprite cyclically through a path that you define, defined by a series of segments. The |ROUTEALL command knows which segment and position each sprite is in and if it reaches a segment change, it modifies the speed of the sprite according to the conditions of the next segment.  
|ROUTEALL does not move the sprite, it only modifies its speed. To move it, it must be used in conjunction with |AUTOALL.

Examples of status byte value assignment:

Typical enemy: a sprite to be printed every cycle, with collision detection with other sprites and animation must have:

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

A house that moves as we move: a sprite that is printed every cycle, but without collision detection and relative movement.

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

A bonus fruit: it is a sprite that is not printed every cycle, but has collision detection.

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

A ship that will follow a predefined trajectory. It will require the path flag, the automatic movement flag, the animation flag, the collision flag and the print flag. This time I'm going to put it in binary directly. As you can see I have set bit 7 to 1, then there are 3 zeros because I have not set the overwrite, collider or relative movement flags and finally I have set 4 active flags corresponding respectively to the automatic movement, animation, collision and printing flags.

$$\text{status}=10001111$$

The sprite attribute table consists of 32 entries of 16 bytes each, starting at address 27000.

The reason for having 16 bytes is none other than performance, since calculating the address of sprite N involves multiplying by 16, which, being a multiple of 2, can be done with a shift. This is useful for operations involving a single sprite. For operations that traverse the sprite table (such as |PRINTSPALL or |COLSP), internally the table is traversed with an index to which 16 is added to move from one sprite to the next. Addition is the fastest in that case.

The attributes that each sprite has are:

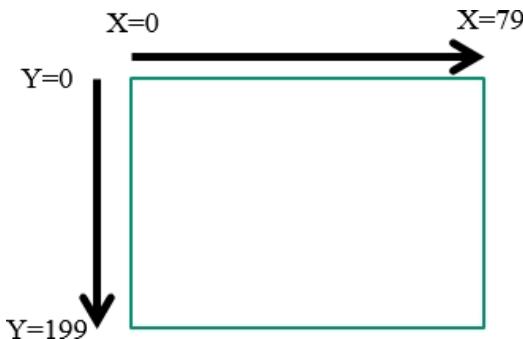
<b>attribute</b>	<b>Byte</b>	<b>Length (bytes)</b>	<b>Meaning</b>
status	0	1	Byte containing the status flags for the PRINTSPALL, COLSPALL, ANIMALL, AUTOALL , MOVERALL, COLSPALL and ROUTEALL operations.
Y	1	2	Coordinate Y [-32768..32768] the values corresponding to the inside of the screen are [0..199].
X	3	2	X-coordinate in bytes[-32768..32768] the corresponding values inside the screen are [0..79].
Vy	5	1	Step to automatic movement
Vx	6	1	Step to automatic movement
Sequence	7	1	Animation sequence identifier [0..31]. If it has no sequence, a zero must be assigned.
Still from	8	1	Frame number in the sequence [0..7].
Image	9	2	Memory address where the image is located
Previous Sprite	10	2	Internal use for the sprite ordering mechanism
Sprite next	12	2	Internal use for the sprite ordering mechanism
Route	15	1	Path identifier to be followed by the sprite

The memory address where the coordinates of each sprite are stored can be calculated as follows:

**Y-coordinate address =** $27000 + 16 \cdot N + 1$  **X-**  
**coordinate address =** $27000 + 16 \cdot N + 3$

Accessing with **POKE** to these addresses we can modify their value, although you can also use **|LOCATESP**.

The 8BP library does not use "pixels" in the X-coordinate, but bytes, so the X-coordinate that falls inside the screen is in the range [0..79]. The Y-coordinate is represented in lines, so the representable range on the screen is [0..200]. If you place a sprite outside these ranges, but part of the sprite is on the screen, the library will do the clipping and paint the part that needs to be seen.



*Fig. 46 Screen coordinates*

The addresses of the attributes of the 32 sprites can be handled with PEEK and POKE, although the animation sequence and path assignment involve more operations and if you want to change them, a POKE is not enough, you have to use **|SETUPSP**. Here is the address list of all the attributes of the 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

Table 3 Attribute addresses of the 32 sprites

The space occupied by each sprite in the table is 16 bytes. As you can see the X and Y coordinates are 2byte numbers. Sprites accept negative coordinates, so you can partially print a sprite on the screen, giving the impression that it is coming in bit by bit. You can't set negative coordinates with POKE, but you can with |LOCATESP and also with |POKE, which is a version of BASIC's POKE command but accepts negative numbers (and 16-bit numbers).

It is good practice to place the character or spaceship at position 31 (there are 32 sprites numbered from 0 to 31). If your spacecraft has position 31 it will be printed last, on top of the rest of the sprites in case of overlap.

## 8.7 All sprites printed and sorted

In the 8BP library you have a command that prints all sprites that have the print flag active at the same time. This is the command |PRINTSPALL

This command has 4 parameters, but you only need to fill them in the first time you invoke it, because the following times, it will remember the parameters, and you only need to fill them in again if you want to change any of them. This is useful because parameter passing is very time consuming (you can save more than 1ms by avoiding parameter passing).

The parameters are:

|PRINTSPALL, <ordenini>, <ordenfin>, <animate>, <synchronism>.

- **The sync parameter** can take the values 0 or 1 and indicates that it will wait for a screen sweep interruption before printing. If you want speed, I don't recommend it. If you want more smoothness, maybe yes.
- **The animation parameter** can take the values 0 or 1. If it is active, before printing each sprite, its animation flag in the status byte will be checked and if it is active, then it will be switched to the next frame. This is very useful with enemies, but with your character you may not, as you may only want to animate him when you move him and not in each frame. **IMPORTANT:** Animation is done before printing, not after printing. That means that if you have just assigned an animation sequence, you will not see the first frame of that sequence.
- **The order parameters ("ordenini", "ordenfin")** indicate the initial and final sprites that define the group of sprites ordered by "Y" coordinate that we are going to print. For example, if we assign the values 0,0 then they will be printed sequentially from sprite 0 to sprite 31. If we assign 0,8 they will be printed from 0 to 8 ordered (9 sprites) and from 10 to 31 sequentially. If we set 0,31 all sprites will be printed in order. If we set a 10,20 the sprites will be printed sequentially from 0 to 9, then they will be printed ordered from 10 to 20 and finally they will be printed sequentially from 21 to 31.

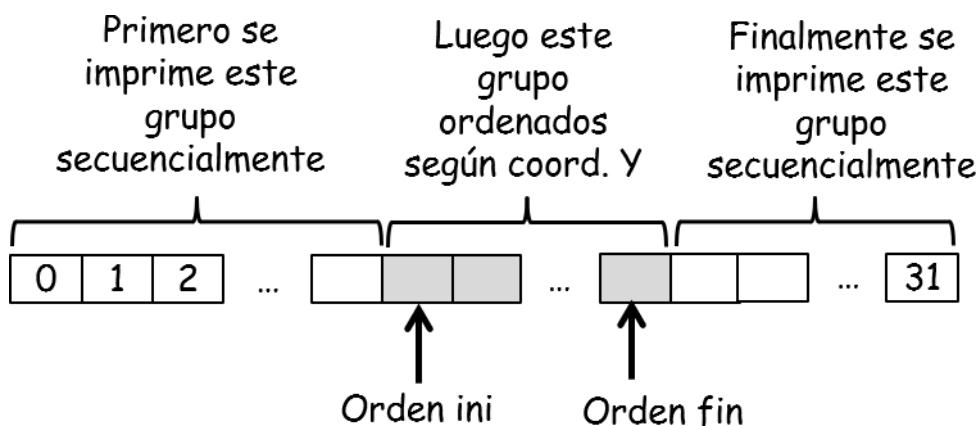


Fig. 47 Sequential and ordered sprite groups

The ordering is very useful for making games like "Renegade" or "Golden AXE", where it is necessary to give a depth effect. The ordering is appreciated when there are overlaps between sprites.



*Fig. 48 Effect of sprite ordering*

- |**PRINTSPALL, 0,0,1,0 : sequentially prints all sprites**
- |**PRINTSPALL, 0,31,1,0 : prints all sprites in orderly fashion**
- |**PRINTSPALL, 0,7,1,0: prints 8 ordered and the rest sequential**
- |**PRINTSPALL, 16,24,1,0: 16 sequential, 9 sorted, 7 sequential**

If the "ordenini" parameter is omitted, the last assigned value is considered, or zero if no value has ever been assigned. Also, if you are going to change either of the two sorting parameters, it is a good idea to first run PRINTSPALL,0,0,0,0 so that the sprites are first reordered sequentially before sorting them with a new configuration.

Printing in order is more computationally expensive than printing sequentially. If you only have 5 sprites that need to be sorted, pass for example a 0.4 as sorting parameters, don't pass 0.31. Sorting all sprites takes about 2.5 ms but if you sort only 5 sprites you can save 2 ms. Maybe you have a lot of sprites and it's not worth sorting some of them, like the shots or sprites that you know won't overlap.

The algorithm used to sort the sprites is a variant of the so-called "bubble" algorithm. Although you will find in the literature that the so-called "bubble" algorithm is very inefficient, this is said by those who talk about a list of random numbers to be sorted. This is a case where the sprites are normally almost ordered and from one frame to the next only one or two sprites are disordered, not more, as their coordinates evolve "smoothly". So the algorithm goes through the list of sprites and when it finds a couple of disordered sprites, it turns them around and stops sorting. It is extremely fast, and even if it is only able to sort a couple of sprites at a time, it is ideal for this use case. Only in the case where there are 2 unordered sprites and they are overlapping, there will be a frame where we see one of them printing out of order, but it will be corrected in the next frame. It is imperceptible.

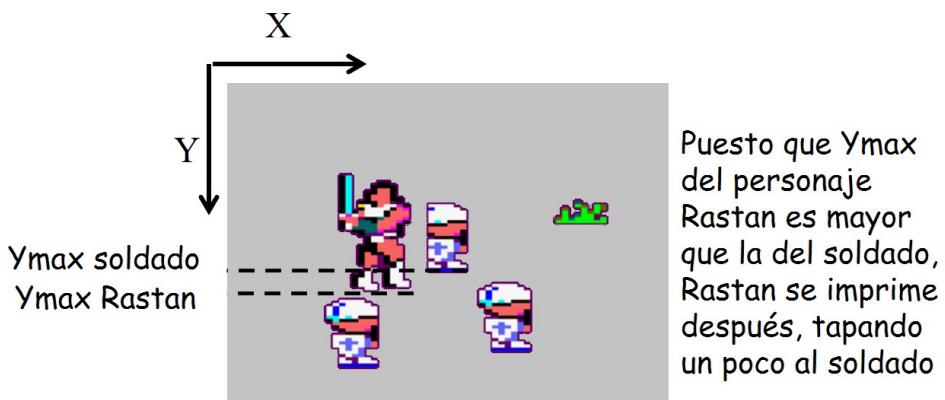
You may sometimes want the sorting to be complete on every frame. That is, not to have a couple of sprites sorted on each invocation of |**PRINTSPALL**, but to be sure that they are all sorted. The **8BP** library allows you to do this through its four sorting modes, which you can set by invoking the command

|The following is a single-parameter **PRINTSPALL** (just run it once to set the sorting mode):

**PRINTSPALL,0 : partial sorting using Ymin** **PRINTSPALL,1 :**  
**complete sorting using Ymin** **PRINTSPALL,2 : partial sorting**  
**using Ymax** **PRINTSPALL,3 : complete sorting using Ymax**

Sorting using Ymax is based on the largest Y-coordinate of the sprites, i.e. where their feet are located rather than their heads. If the sprites are the same size, a Ymin-based sorting may work, but if the sprites are of different heights you may want to sort according to where each character's feet are, and for that you will have to use sorting mode 2 or 3.

Ymax sort modes are slower, about 0.128 ms per sprite, so use them when you really need them.



*Fig. 49 Sorting according to Ymax*

The full sort consumes very little more than the partial sort (about 0.3ms). This is because the sprites are hardly ever jumbled from one frame to the next, but even those 0.3ms are worth saving if possible.

Remember that the PRINTSPALL command has "memory", so that it is enough to invoke it the first time with parameters and from that moment on we can invoke PRINTSPALL without parameters because the command "keeps" the values of the parameters with which it was invoked and there is no need to pass them to it unless they change. This saves more than 1ms, since the parser works less.

## 8.8 Collisions between sprites

To check if your character or your shot has collided with other sprites you can use the command

|COLSP, <sprite\_number>, @collision%.

Where sprite number is the sprite you want to test (your character or your shot) and the variable "collision" is an integer variable that previously had to be defined, assigning an initial value, for example:

**collision%=0**

### **|COLSP, 1, @collision%, 1, @collision%, 1, @collision%.**

The variable "collision" will be filled with the first sprite identifier that is detected to have collided with your sprite, although multiple collisions may occur, but the command only gives you one result.

Internally, the 8BP library goes through the colliding sprites from 31 to 0 (it goes through them in reverse order), and if they have the collision flag active (bit 1 of the status byte) then it checks if it collides with your sprite. If there is no collision with any of them, the collision% variable is set to 32. If there is, it will return the number of the sprite that is colliding with your sprite. If for example 4 and 12 collide, the function will return a 12 because it checks 12 before 4.

**Neither your character nor your shot must have both the "collider" flag (bit 1) and "collider" (bit 5) active at the same time, otherwise they will always collide... with themselves! That is, a sprite cannot have bits 1 and 5 active at the same time.**

Collision between sprites is an expensive task. Internally the library needs to calculate the intersection between the rectangles containing each sprite to determine if there is overlap between them. To save calculations, it is best to place enemies in consecutive sprite positions. If for example the enemies we can collide with are sprites 15 to 25, we can set the collision to check only those sprites. To do this, we invoke collision on sprite 32, which does not exist. This will tell the 8BP library that this is configuration information for the command, indicating the **range of collisable sprites to be scanned for each collider**:

### **|COLSP, 32, <sprite initial>, <sprite final>.**

Example:

**|COLSP, 32, 15, 25**

This optimisation is not very significant, but it becomes so when COLSP is invoked several times or when the |COLSPALL command is used, which internally invokes COLSP several times.

Another interesting optimisation, capable of saving 1 milliseconds in each invocation, is to tell the command to always use the same BASIC variable to leave the result of the collision. To do this, we will indicate it using 33 as the sprite, which does not exist either.

**col%=0**

**|COLSP, 33, @col%, @col%, @COLSP, 33, @col%, @COLSP, 33, @col%**

Once these two lines have been executed, subsequent invocations of COLSP will leave the result in the variable col, without the need to indicate it, for example:

**|COLSP, 23 : REM this invocation is equivalent to |COLSP, 23, @col%.**

**IMPORTANT:** The collision variable in the |COLSP command is not the one used in the |COLSPALL command. They are different variables (unless you pass both commands the same variable to act on it).

## 8.9 Adjusting sprite collision sensitivity

It is possible to adjust the sensitivity of the COLSP command, deciding whether the overlap between sprites must be several pixels or only one pixel, in order to consider that a collision has occurred.

This can be done by setting the number of pixels (pixels in Y direction, bytes in X direction) of overlap needed in both Y and X direction, using the COLSP command and specifying sprite 34 (which does not exist).

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

The 8BP library does not use "pixels" in the X-coordinate, but bytes, so you must take into account that a 1-byte collision is actually 2 pixels and that is the minimum possible collision when you set dx=0.

In the Y-coordinate, the library works with lines so that dy=0 means a single pixel collision.

A strict collision, useful for shooting, would be one that does not tolerate any margin, considering collision as soon as there is a minimum overlap between sprites (1 pixel in Y direction or one byte in X direction).

|COLSP, 34, 0, 0, 0: rem collision as soon as there is a minimum overlap

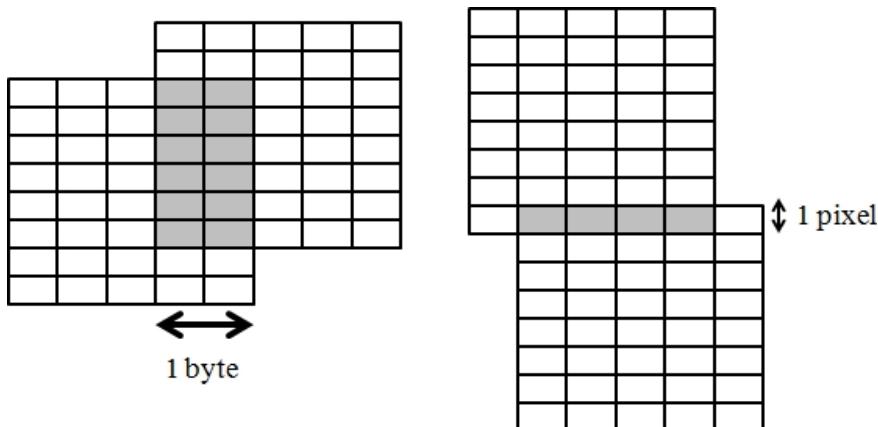


Fig. 50 Strict collision with COLSP, 34, 0, 0

However, if we are making a game in MODE 0, where the pixels are wider than tall, it is perhaps more appropriate to give some margin on the Y axis, but none on the X axis. For example:

|COLSP, 34, 2, 0 : rem collision with 3 pix in Y and 1 byte in X

My recommendation is that, if there are narrow or small shots, set the collision to (dy=1, dx=0) while if there are only large characters you can leave it with more margin (dy=2, dx=1). You should also consider that, if your sprites have an erasure "margin" around them to move around erasing themselves, that margin should not be part of the collision consideration so it makes sense that both dy and dx should be non-zero. In any case, this is something you will decide depending on the type of game you make.

## 8.10 Who collides and with whom: COLSPALL

With the **|COLSP** function we have seen so far, collision detection of one sprite with all the others is possible. However, if we have a multiple shot, where for example our ship can fire up to 3 shots simultaneously, we would have to detect the collision of each of them and additionally that of our ship, resulting in 4 invocations to **|COLSP**.

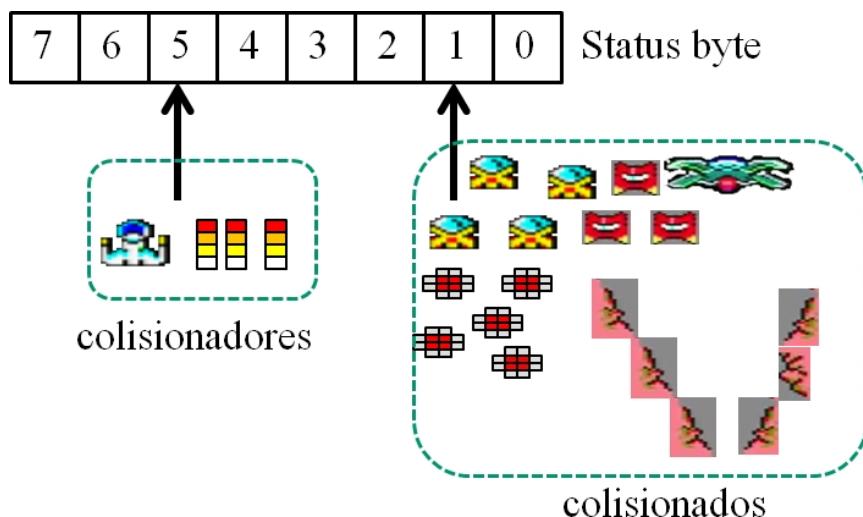
Keep in mind that each invocation goes through the parsing layer, so four invocations are costly. For this we have a very powerful command: **|COLSPALL**.

This function works in two steps: first we must specify which variables are going to store the collider and the collided sprite. The following instruction will be executed only once, and serves to define the variables on which we will obtain the results, which must exist previously:

**|COLSPALL, @collider%, @collider%, @collider%**.

And subsequently, in each game cycle we simply invoke the function **|COLSPALL** without parameters.

The function will consider as "collider" sprites those that have the collider flag set to "1" in the status byte (bit 5), and as "collided" those sprites that have the collision flag (bit 1) of the status byte set to "1". The colliding sprites should be our ship and our shots, and the colliding sprites should be all those with which we can collide: enemy ships and shots, mountains, etc. As I said before, a sprite must not have both bits active (=1) at the same time.



*Fig. 51 Colliders versus colliders*

The **|COLSPALL** function starts by checking sprite 31 (if it is a collider) and works its way down to sprite 0, internally invoking **|COLSP** for each collider sprite. For each collider, the collider sprites are also traversed in decreasing order (from 31 to 0). As soon as it detects a collision, it interrupts its execution and returns the value of the collider and the collider. It is therefore important that our ship has a higher sprite than our shots. That way, if we are hit

we will detect it, even if we have hit an enemy with a shot at the same instant.

In each game cycle only one collision can be detected, but that is enough. It is not a major limitation that in each frame only one enemy can start to "explode". If, for example, you throw a grenade and there is a group of 5 soldiers affected, each soldier will start to die in a different frame, and after 5 frames they will all be exploding. Using |COLSPALL won't blow them all up at once, but your game will be faster, and in an arcade game that's very important.

In case of invoking **|COLSPALL** with a single parameter,  
**|COLSPALL, <initial collider>**.

Colliders will be scanned from the indicated collider -1 to sprite zero, in descending order. So if you need to detect more than one collision per game cycle, you can do so by successively invoking **COLSPALL, <collider>** until the collider variable takes the value 32.

**Example:**

**|COLSPALL, 7 : rem searches for collisions from collider 6**

**8.10.1 How to programme a multiple shot using COLSPALL** The first thing you need to do is to decide how many active shots can be fired. If you decide that your ship can fire 3 projectiles at a time, then you should reserve 3 sprite identifiers to trigger. Next, you must adjust the delay between one shot and the next to prevent two projectiles from almost hitting each other if you shoot too fast. This can be done by defining a minimum delay between shots. In the following example I have set a delay between shots of 10 game cycles. To do this, pressing the space bar (key 47) checks if at least 10 cycles have elapsed since the last shot. If not, it will not fire.

```
130 ----- 'cycle of play'.
150 |AUTOALL,1:|PRINTSPALL,0,1,0
170 'character movement routine -----
172 IF INKEY(47)=0 THEN IF delay<cycle-10 THEN delay=cycle: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:'go right
190 IF INKEY(34)=0 THEN dir=1:|SETUPSP,0,6,-1:'go left
193 cycle=cycle+1
310 GOTO 150
```

To choose the sprite to be used as a trigger, the instruction is executed:

**disp = 1+ disp Mod 3**

This will allow your shot to take the values 1,2,3,4 alternatively. If you need the sprites to be 20,21,22,23 you can use **disp = 20 + disp Mod 3** .

because if you put 21 as the sum, it doesn't work (try it yourself). That's the thing about modular arithmetic.

And now we come to collisions and the advantage of using COLSPALL, much faster than invoking COLSP multiple times. The only important recommendations are:

- Let our <sprite number> be higher than our triggers, so that |COLSPALL check it before shooting.
- That we have configured |COLSP to only check the list of sprites that are enemies and are required to collide, by using |COLSP 32, <start>, <end>.

Before starting the game cycle we define our variables:

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

In the game cycle we will put:

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

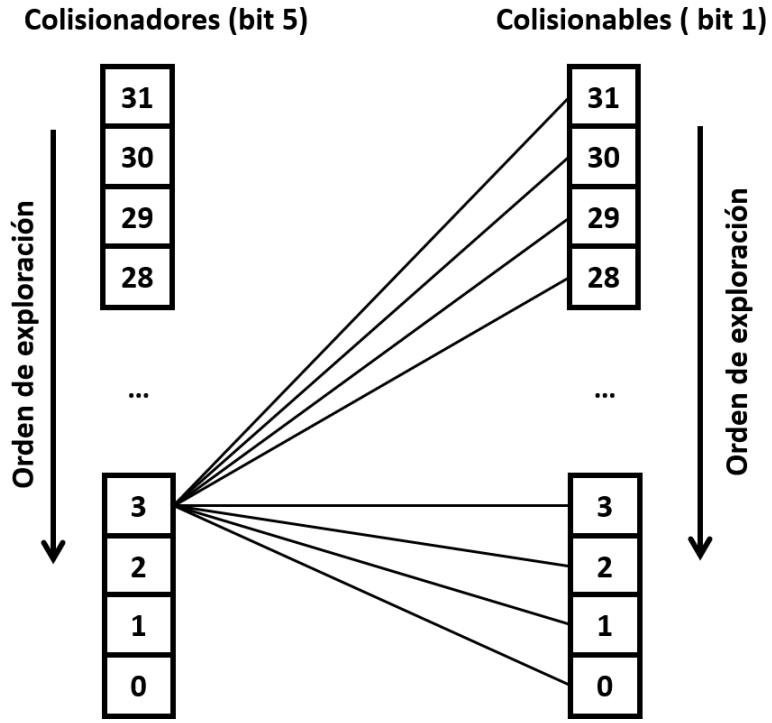
With this line we already know if there is a collision, because then the variable "collider" will be <32 Furthermore, if it is equal to 31 then it is our ship (we have been hit) and if not, then for sure one of our shots has hit an enemy ship and we will go to the routine located in line 770, where we will find something like this:

```
769' --- routine collision shot -----  
770 |SETUPSP, collider, 9, img deleted:'associate deleted image to the  
shot  
772 |PRINTSP, collider: 'we delete the shot  
775 |SETUPSP, collider, 0, 0: 'deactivate triggering  
777 if collided>=duros then return:'enemy indestructible  
778 ' sequence 4 is an animation sequence of 'Death', a  
explosion  
780 |SETUPSP, collided, 7, 4:|SETUPSP, collided, 0, &x101: return
```

In short, with a single invocation of COLSPALL we know who has collided ("collider"), and with whom they have collided ("collided").

### 8.10.2 Who collides when there are several overlaps

It is very important to keep in mind that 8BP goes through the colliders from 31 to 0 and for each of them, it goes through the colliderables from 31 to 0. We must associate the Sprite IDs to our sprites depending on how we want them to collide.



*Fig. 52 collision check order*

If our sprite (collider) collides with two sprites in the same area, we can know a priori which of them we are going to collide with, which is very useful for certain games.

Let's assume the game "frogger", in which a frog must cross a river by jumping over logs. If the collision is on a log, we will not die, but if the collision is on a river, we will die.

To program it we can put 4 rivers (4 immobile elongated sprites) and on them move some sprites that are trunks. We could set up the following distribution:

- The frog is sprite 31 (assume it is a collider).
- The trunks are sprites 4, 5, 6, 7 (collidable).
- The rivers are sprites 0, 1, 2, 3 (collidable).

Rivers can have the print flag disabled, so they can collide with the frog (collided flag) without being printed. That is, we would set them status =2



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



*Fig. 53 in case of overlapping we are interested in the collision of the trunk.*

Well, as the logs and the river overlap, the moment the frog climbs on a log it collides with both, but the log is checked first as it has a higher ID sprite. The collision command will detect only the collision with the log. On the contrary, if the frog jumps over the water, then the collision command will detect the river and after evaluating from **BASIC** the variable "collided" and seeing that it is a river, we would determine that our frog must die.

### 8.10.3 Advanced use of the status byte in collisions

Sometimes you may want an enemy not to kill you by colliding with your character because they are in a special state, or because they are simply far away in a game that pretends enemies are approaching.

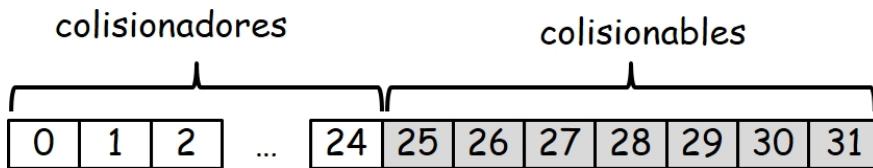
Some special circumstances may require the use of a "mark" on the sprite to indicate that, although there has been a collision, it should not die, or should not kill you.

For this you can use the flags that you don't use in the sprite and check them in your collision routine.

Let's see an example for an enemy that we want to make harmless because it is far away (simulating 3D) or with low energy but collides with us.

Suppose your character is collided and the enemy is a collider. You can force that collision is only detected with sprites bigger than 25 and if the enemy is number 20 (for example) it won't be able to collide with itself.

|COLSP,32,25,31



*Fig. 54 effect of COLSP,32,25,31*

As a "harmless" indicator we will use the collided flag, setting it to 1. So we will set the enemy (sprite 20) flag to 1 in its status byte.

Now suppose the |COLSPALL command detects a collision, and leaves the result in collider and collided.

```

|COLSPALL
If collider<32 then GOSUB 100
<instructions>.

100 rem collision routine
110 dir=27000 + collider*16 :rem byte address status of collider
120 if PEEK (dir) and 2 THEN RETURN: rem harmless if bit collided=1
130 <an enemy has collided that is not harmless>

```

The "**if PEEK (dir) and 2**" instruction is the one that checks the collided bit since 2 in binary is 00000010, just the position of that bit in the sprite state.

When the enemy is no longer harmless we simply set its collided flag to 0 and upon further collision, it will kill us.

This technique is perfectly valid using any other unused flag.

## 8.11 Table of animation sequences

Animations are usually composed of an even number of frames, although this is not a strict rule. Think for example of a simple character animation with only two frames: legs open and closed. That's two frames. Now think of an enhanced animation, with an intermediate movement phase. This means creating the sequence: closed-intermediate-open-open-intermediate-and start again. As you can see it's an even number, that's 4

8BP animation sequences are lists of 8 frames, they cannot have more, although you can always make shorter sequences.

The frames of an animation sequence are the memory addresses where the images of which they are composed are assembled, and they can be different in size, although normally they are the same. If in the middle of the sequence you enter a zero, the meaning is that the sequence has ended.

Although before V33 there was an RSX command called **|SETUPSQ** to create sequences from BASIC, I have removed it in V33, because its use is more complex than defining sequences from the sequences.asm file and, in fact, I have never used the **|SETUPSQ** command in any of my games, so I decided to sacrifice it to save memory.

Let's look at an example of creating a sequence in the sequences.asm file.

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

Before explaining how to assign a sequence to a sprite, let me remind you how to assign images to sprites: Since version V26 of 8BP, there is the possibility to include a list of images (their labels) in a list called IMAGE\_LIST in your images\_your\_game.asm file. With this you can reference the images from BASIC with an index instead of a memory address. This way you don't have to look up memory addresses every time you assemble. This applies to the instruction:

**|SETUPSP, #, 9, <address>.**

The example shows an animation sequence of 3 different frames, but to make it smooth before starting again you have to go through the "middle" frame again (note that the second and fourth frames are the same), so in the end there are 4 frames:



and back to square  
one

*Fig. 55 Animation sequence*

If you wanted to make a sequence of more than 8 frames you could simply chain two sequences in a row and when the character reached the last frame of the first sequence use the **|SETUPSP** command to assign the second sequence to it.

Animation sequences are assembled from address 33600 and you can define up to 31 animation sequences (from number 32 onwards they are not considered sequences, but "macro sequences", which is another concept). Each sequence will be identified by a number in the range [1..31]. The sequence zero does not exist, it **is used to indicate that a sprite has no sequence**.

To assign a sequence to a Sprite use the SETUPSP command with parameter 7:

<b> SETUPSP, &lt;sprite_id&gt;, 7, &lt;sequence number&gt;</b>
--

With this command, the animation sequence is assigned to the sprite in the corresponding field of the sprite table, and a zero is placed in the frame ID field. In addition, the corresponding image is assigned to the first image of the sequence. If you are using |ANIMALL before printing or |PRINTSPALL with animation flag, even if SETUPSP places the animation at frame zero, you will jump to frame 1 before printing. This is normally not a problem, but in the case of a "death sequence" (more on this later) where for example the first frame is to erase the sprite, you may not want to jump directly to frame 1.

1. In that case, a simple trick can be to repeat frame zero in the definition of the death sequence. That way you make sure that the frame is visible. Another option is to remove the animation flag and animate it with |ANIMASP after printing.

Each sequence stores 8 memory addresses corresponding to the 8 frames, that is 16 bytes consumed by each sequence.

Your animation sequence file may look something like this:

```
;=====
; up to 31 animation sequences
=====
must be a fixed table and not a variable table
; each sequence contains the addresses of cyclic animation frames
Each sequence is 8 image memory addresses.
even number because the animations are usually an even number
a zero means end of sequence, although 8 words are always spent.
; by sequence
When a zero is found, a new start is made.
if there is no zero, after frame 8 it starts again.
If a Sprite is assigned the sequence zero, it has no sequence.
We start from sequence 1
----- character animation sequences of the character montoya-----
SEQUENCES_LIST
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0,0 ;1
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0,0 ;2
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0,0 ;3
dw MONTOYA_UL0,MONTOYA_UL1,MONTOYA_UL2,MONTOYA_UL1,0,0,0,0,0 ;4
dw MONTOYA_L0,MONTOYA_L1,MONTOYA_L2,MONTOYA_L1,0,0,0,0,0 ;5
dw MONTOYA_DL0,MONTOYA_DL1,MONTOYA_DL2,MONTOYA_DL1,0,0,0,0,0 ;6
dw MONTOYA_D0,MONTOYA_D1,MONTOYA_D0,MONTOYA_D2,0,0,0,0,0 ;7
dw MONTOYA_DR0,MONTOYA_DR1,MONTOYA_DR2,MONTOYA_DR1,0,0,0,0,0 ;8

----- soldier animation sequences ----- dw
SOLDIER_R0,SOLDIER_R2,SOLDIER_R1,SOLDIER_R2,0,0,0,0,0 ;9
dw SOLDIER_L0,SOLDIER_L2,SOLDIER_L1,SOLDIER_L2,0,0,0,0,0 ;10
```

## 8.12 Special animation sequences

Various types of animation sequences are available in 8BP:

Type of sequence	description
Normal Sequence	The sequence of animation frames is repeated over and over again. They end either in a picture direction or in a

	a zero if we want to make a sequence of less than 8 images
<b>Death sequence</b>	The last frame of the sequence is a "1". This tells 8BP to change the state of the Sprite to zero. Normally the frame before the "1" is an erase image.
<b>End sequence</b>	After running through the sequence the Sprite runs out of animation. The last frame of the sequence is a "2". This tells 8BP to remove the animation flag from the Sprite state.
<b>Chained sequence</b>	After traversing the sequence, the last frame indicates the identifier of the next sequence that will be assigned to the Sprite. Using this type of sequences you can build animation sequences longer than 8 frames.
<b>macro-sequences</b>	They allow to associate a sequence according to the speed of the Sprite, automatically.

### 8.12.1 Death sequences

The 8BP library allows you to make "death sequences", which are sequences where the sprite goes to an inactive state at the end of the sequence. This is indicated by a simple "1" as the value of the memory address of the final frame. These sequences are very useful for defining explosions of enemies that are animated with |ANIMA or |ANIMALL. After hitting them with your shot, you can associate a death animation sequence to them and in the following game cycles they will go through the different animation phases of the explosion, and when they reach the last one they will go to inactive state, not printing any more. This inactive state is done automatically, so what you have to do is simply check the collision of your shot with the enemies and if it collides with any of them you change the state with SETUPSP so that it can't collide any more and assign it the animation sequence of death, also with SETUPSP.

If you use a death sequence, don't forget to make sure that the last frame before you find the "1" is a completely empty one, so that there is no trace of the explosion.

Example of a death sequence (note that it includes a "1"):

```
dw EXPLOSION_1,EXPLOSION_2,ExPLOSION_3,1,0,0,0,0,0
```

An interesting effect is to cycle through several frames repeatedly before ending with a black frame that serves to erase

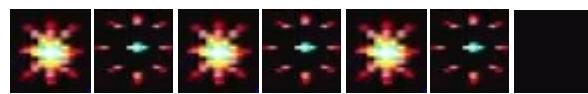


Fig. 56 Death sequence

The "1" now appears in eighth position:

```
dw EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2,  
ExPLOSION_3,1
```

Remember that if you use the |PRINTSPALL command with the animation flag active, it is animated first and then printed, so the first image of the death sequence will not be seen. A simple trick to get it to show is to repeat it twice.

### 8.12.2 End sequences

End sequences have existed since version V42. They allow to associate an animation sequence to a Sprite that we want to be carried out only once. At the end of the sequence, the Sprite will have no animation flag. The other flags of its state are respected.

We simply need to put a "2" in the last frame of the sequence.

Here are two examples. As soon as the sequence reaches frame "2", the sprite is no longer animated.

```
SEQUENCES_LIST
dw ;1
MONTOMYA_R0,MONTOMYA_R1,MONTOMYA_R2,MONTOMYA_R1,2,0,0,0,0 ;2
dw IMG1, IMG2, IMG3, IMG4, IMG5, IMG6, IMG7,2
```

### 8.12.3 Chained sequences

Chained sequences have existed since version 8BP V42 . They allow sequences longer than 8 frames to be built by chaining sequences together.

The mechanism is simple. Simply the last frame of the sequence must be the number of the sequence you want to assign next.

As you can imagine, you will not be able to assign sequence "1" or "2", since those numbers mean "die" or "end". That is, you will be able to chain sequences starting from number 3.

In this example I have chained sequences 4 and 5 so that after one, the other is assigned and vice versa. It's like having a sequence of 14 frames. We could chain more sequences and make the animation much longer. Each sequence we add is 7 more frames (not 8) because we need the last one to indicate the next sequence.

You can also make them shorter and fill with zeros up to 8 frames, but the sequence number must appear before any zero, because when a zero is encountered the animation cycles.

```
_SEQUENCES_LIST
dw MONTOMYA_R0,MONTOMYA_R1,MONTOMYA_R2,MONTOMYA_R1,0,0,0,0,0 ;1
dw MONTOMYA_UR0,MONTOMYA_UR1,MONTOMYA_UR2,MONTOMYA_UR1,0,0,0,0,0 ;2
dw MONTOMYA_U0,MONTOMYA_U1,MONTOMYA_U0,MONTOMYA_U2,0,0,0,0,0 ;3
dw IMG11, IMG2, IMG3, IMG4, IMG5, IMG6, IMG7, 5 ;4
dw IMG18, IMG9, IMG10, IMG11, IMG12, IMG13, IMG14, 4 ;5
```

### 8.12.4 Animation macro-sequences

This is an "advanced" feature available from version V25 of the 8BP library. A "macro sequence" is a sequence made up of sequences. Each of the

constituent animation sequences is the animation to be performed in a particular direction. The direction is determined by the speed attributes of the sprite, which are in the sprite table. Thus, when we animate a sprite with |ANIMALL, it will automatically change its animation sequence without us having to do anything (you don't actually need to invoke |ANIMALL because |PRINTSPALL already does this internally if you set a parameter).

Macro sequences are numbered starting at 32. It is very important to place the sequences within the macro sequence in the correct order, i.e. the first sequence should be for when the character is still, the next for when the character goes left ( $Vx < 0$ ,  $Vy = 0$ ), the next for right ( $Vx > 0$ ,  $Vy = 0$ ), etc, in the following order (be careful because it is easy to make a mistake):



*Fig. 57 Order of sequences in a macro sequence*

If the sequence assigned to the still position is zero, then it is simply animated with the last assigned sequence.

Macro sequences must be specified in the file sequences\_yourgame.asm, an example of which is given below:

```
;-----  
; animation sequences  
;  
;  
_SEQUENCES_LIST  
dw NAVE,0,0,0,0,0,0,0,0;1  
dw JOE1,JOE2,0,0,0,0,0,0,0;2 UP  
JOE dw JOE7,JOE8,0,0,0,0,0,0,0;3  
DW JOE dw  
JOE3,JOE4,0,0,0,0,0,0,0;4 R JOE dw  
JOE5,JOE6,0,0,0,0,0,0,0;5 L JOE  
  
_MACRO_SEQUENCES  
;-----MACRO SEQUENCES -----  
are groups of sequences, one for each direction. the meaning is:  
; still, left, right, up, up-left, up-right, down, down-left, down-right  
The numbers are numbered from 32 onwards  
db 0,5,4,2,5,4,3,5,4;sequence 32 contains the sequences of the soldier Joe
```

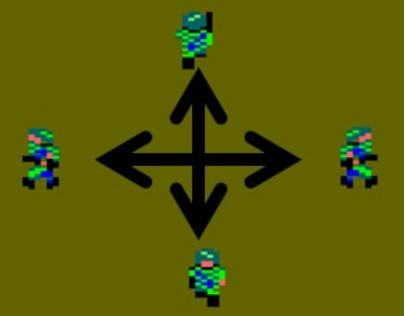
With this sequence definition we can make a simple game that allows us to move "Joe" around the screen without controlling his animation sequence. We assign the sequence 32 and altering the speed, the |ANIMA command (invoked from within the |PRINTSPALL) is in charge of changing the animation sequence if its speed denotes a change of direction. To move the sprite we need to invoke |AUTOALL, since pressing the controls does not change its coordinates but its speed and |AUTOALL will update the sprite's coordinates according to its speed.

<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,31,0,0,&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 'begins game cycle
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

```



Note that I have not defined the sequence for when the character does not move. In that position I have put a zero in the macro sequence. That means that, if the character starts still, you don't know which sequence to assign as there is no "last" sequence used. That's why I assign sequence 2 before assigning 32, so I make sure that the character already has a sequence, even if it is standing still.

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



## 9 Your first simple game

You now have the knowledge to try a first step in the creation of video games. To do this, let's look at a simple example of a soldier that you are going to control, making him walk left and right on the screen.

Let's suppose that we have edited a soldier, thanks to SPEDIT. And we have built its animation sequences, which have been defined in the file "**sequences\_mygame.asm**" and have been left with the identifier 9 and 10 for the left and right movement directions respectively.

The two animation sequences have been created from the sequences.asm file.

```
10 MEMORY 24499
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restore default palette just in case'.
26 ink 0,0:'black background
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' set the limits of the game screen
50 x=40:y=100:' character coordinates 51|SETUPSP,0,0,1:'character status 52|SETUPSP,0,7,9:'animation sequence
assigned to start 53|LOCATESP,0,y,x:'place the sprite (without
printing it yet)

60 'game cycle
70 gosub 100
80 |PRINTSPALL,0,0
90 goto 60

99 ' character movement routine -----
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
```

With this list you already have a mini-game that allows you to control a soldier and make him run horizontally. Note that, if when walking to the left you exceed the minimum value of the limit set with |SETLIMITS, the character will be "clipped", showing only the part that is inside the allowed play area.

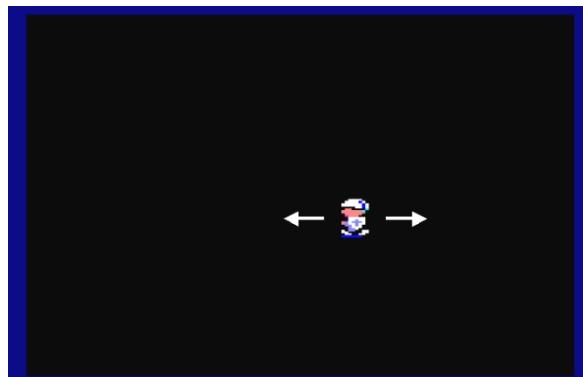
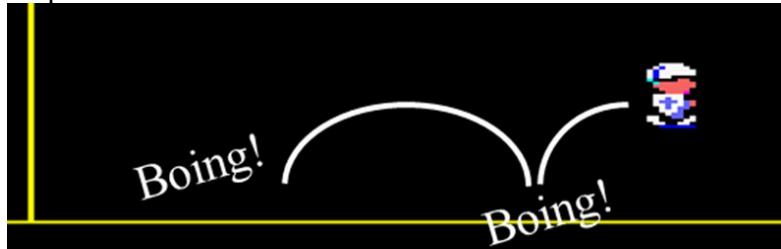


Fig. 58 A simple game

### 9.1 Now, let's jump! Boing, boing!

In the example above our character only moves from left to right. If we want to program a jump, we can do so by storing the vertical trajectory in a

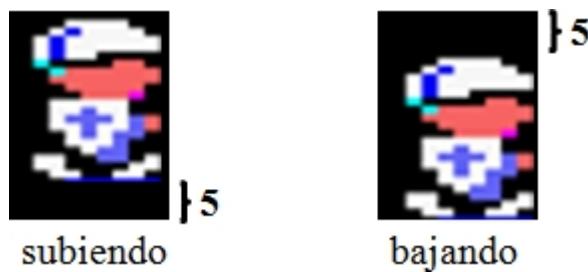
BASIC array. Later we will see a better way to do it (with 8BP paths) but for now it will serve as an example.



*Fig. 59 our character can jump*

The jump trajectory is defined for the Y-coordinate. First it goes up 5 lines at once, then up 4, then up 3, etc. until it reaches zero. At that point the direction of movement is reversed and we start going down 1 line, then 2, then 3, etc. up to 5.

So that when the dummy goes up and down it does not leave a trace, we will have to have a drawing of the dummy going up with 5 black lines underneath to erase itself when going up and in the same way, another image with 5 black lines above to go down. In this case they are the images 22 and 23 to jump to the right and 24,25 to the left.



*Fig. 60 Up and down image*

At the zenithal point of the jump, the up image must be changed to the down image, but first we must go up 5 lines at once, because if you compare both images, you will realise that if you go from one to the other directly, it is like going down 5 lines.

```

10 MEMORY 24999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 ON BREAK GOSUB 2800
30 CALL &BC02:'restore default palette just in case'.
40 INK 0,0: 'black background
50 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
80 |SETLIMITS,12,80,0,186: ' we set the screen limits
90 x=40:y=100:' coordinates of the character
100 |SETUPSP,0,0,1:' character status
110 |SETUPSP,0,7,9:'Animation sequence assigned on start
120 |LOCATESP,0,y,x:'place the sprite (without printing it yet)
121 DIM jump(24):' jump data
122 for i=-5 to 5: k=k+1:skip(k)=i: k=k+1: skip(k)=i: next:
skip(11)=-5: skip(23)=5
125 PLOT 1,150:DRAW 640,150: plot 92,150:draw 92,400:'floor and wall
126 |MUSIC,0,0,5: 'music starts playing
130 ----- 'cycle of play'.
150 |LOCATESP,0,y,x:|PRINTSPALL,0,0
151 GOSUB 170
160 GOTO 130:' end of game cycle

```

```

170 ' character movement routine -----
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 jumping routine -----
270 IF jump=11 THEN |SETUPSP,0,9,DIR*2+23 ELSE IF jump=23 THEN
y=y+jump(jump):jump=0:|SETUPSP,0,7,DIR+9:return
280 y=y+jump(jump)
300 jump=jump+1
310 return
2800 |MUSIC:MODE 1: INK 0,0: PEN 1

```

As you can see in the list, if you press the "Q" key, the variable "jump" is equal to 1 and at that moment the logic of the dummy becomes complicated because it requires executing an IF statement to change the image when it reaches the zenith point and it is also necessary to update the Y coordinate of the dummy and the variable "jump".

Later on we'll look at how to do this with a more advanced technique, using sprite "routes". **8BP's programmable routes provide a more efficient method of doing this sort of thing, so you'll see your character jump much faster.** Routes will allow you to execute a trajectory (a jump, a circle, etc.) without having to check the coordinates at every instant. And you can also change the state of a sprite in the middle of a route, or change its associated image, its sequence or even change its route, concatenating different routes.



## 10 Screen sets: layout or "tile map".

### 10.1 Layout Definition and Use

Often you will want your games to consist of a set of screens where the character must collect treasure or dodge enemies in a maze. In such cases it becomes indispensable to use a matrix where you define the constituent blocks of each "maze" or so-called "layout" of the screen. Sometimes this concept is also called a "tile map" (a "tile" is the English word for "tile").

In the **8BP** library you have a simple mechanism to do this, which also provides a collision function for you to check if your character has moved into an area occupied by a "brick". This mechanism is called "layout". In 8BP a layout is defined by a matrix of 20x25 "blocks" of 8x8 pixels, which can be occupied or not. That is, there are as many blocks as there are characters on the screen in mode 0.

To print a layout on the screen you have the command:

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

This routine prints a row of sprites to build the layout or "maze" of each screen. The matrix or "layout map" is stored in an area of memory that handles 8BP so that when you print blocks you **are not only printing on the screen, but you are also filling the area of memory occupied by the layout** (20x25 bytes) where each byte represents a block.

The y,x coordinates are passed in character format, i.e. y  
takes values [0,24].  
x takes values [0,19].

The blocks printed by the |LAYOUT function are constructed with character strings and each character corresponds to a sprite that must exist. Thus the "Z" block corresponds to the image assigned to sprite 31. The "Y" block corresponds to the image assigned to sprite 30, and so on.

The sprites to be printed are defined with a string, whose characters (32 possible) represent one of the sprites following this simple rule, where the only exception is the blank space representing the absence of a sprite.

Character	Sprite id	ASCII code
" "	NONE	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
"T"	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

*Table 4 Character and Sprite mapping for the |LAYOUT command*

The @string is a string variable. You cannot pass the string directly. That is, it would be illegal to do something like:

**|LAYOUT, 1, 0, "ZZZ YYY".**

The correct is:

**String\$ = "ZZZ YYY".**

**|LAYOUT, 1, 0, @string\$**

Be careful that the string is not empty, otherwise the computer may crash! In addition, you must prefix the string variable with the "@" symbol in order to

that the library can go to the memory address where the string is stored and traverse it, printing the corresponding sprites one by one.

You should note that blanks mean no sprite, i.e. nothing is printed in the positions corresponding to the blanks. If there was previously something in that position, it will not be erased. If you want to erase you need to define an 8x8 erase sprite, where everything is zeros.

Although you use the sprites to print the layout, right after printing it you can redefine the sprites with |SETUPSP and assign them images of soldiers, monsters or whatever you want, that is, the layout "relies" on the sprite mechanism to print, but it does not limit the number of sprites, because you have all 32 sprites to be whatever you want right after printing the layout.

To detect collisions with the layout you have the COLAY function, which can be used with a variable number of parameters.

```
|COLAY,<ASCII threshold>, @collision , <sprite number>.  
|COLAY, @collision , <sprite number>  
|COLAY, <sprite number>, <sprite number>.  
|COLAY
```

Given a sprite and depending on its coordinates and size, this function will find out if it is colliding with the layout and will warn you through the collision variable, which must be previously defined.

The **<ASCII threshold>** parameter is optional and allows the command not to consider collisions for ASCII codes below this threshold. By default it is 32 (which corresponds to the white space). To understand this, it is necessary to take into account the correspondence between ASCII values and Sprites shown in the previous table. For example, if we set the threshold to 69 ("E" code, sprite 10), then sprites 9, 8, 7, 6, 5, 4, 3, 2, 1, and 0 will not be "collisable", so if our character passes over them, the collision will simply not be detected.

It is only necessary to invoke COLAY with the threshold parameter once, as subsequent invocations already take this threshold into account.

Example of use:

```
col%=0  
|COLAY, @col%, 20: REM this is an example with spriteID=20
```

If you invoke the COLAY command without parameters, it will consider the last ones it used. That way you can save the parameter passing and speed up the command by 0.5 ms.

If there is no collision, the variable will take the value zero. Collision occurs if the sprite collides with any layout element other than whitespace (" "), whose ASCII code is 32. If the threshold is used, collision occurs if the layout element has an ASCII greater than the threshold you define.

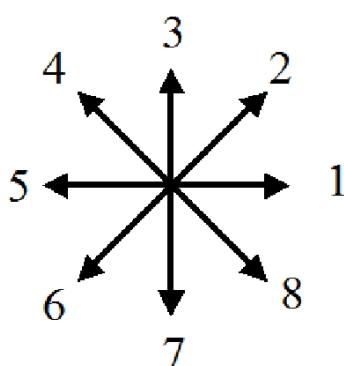
We are going to see an example of creating a layout and moving a character within the layout, correcting its position if it has collided.

Two last considerations on the **COLAY** command:

- The **|COLAY** command is not affected by the sprite collision sensitivity setting (configurable with **|COLSP,34, dy, dx**). The collision sensitivity setting only affects the **|COLSP** and **|COLSPALL** commands.
  - The **|COLAY** command does not take into account the actual size of the images used as "Tiles" or "blocks" of the layout. That is, it considers all blocks specified in the **|LAYOUT** command to be 8x8 pixels of mode 0 (4 bytes x 8 lines), even if you put larger images.

## **10.2 Example of a game with layout**

We are going to evolve the gameplay presented in the previous chapter a bit, as far as character control is concerned. This time we are going to use Montoya as an example, who has 8 animation sequences, each one to move in a different direction. The animation sequences have been assigned a number from 1 to 8.



*Fig. 61 Use of the layout in a game*

In the character control routine we have included collision with the layout. Depending on the direction in which we move, we modify the "new" coordinates ( $yn$ ,  $xn$ ) and call the collision function with layout |COLAY,0 to check if sprite 0 (our character) has collided. If it has collided, we correct the coordinates (one or both) to leave it in a non-collision position before printing it again.

```
10 MEMORY 24999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 CALL &bc02:'restore default palette just in case'.
26 ink 0,0:'black background
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,0,80,0,200: ' set the limits of the play screen
50 dim c$(25):for i=0 to 24:c$(i)=" ":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
110 c$(2)= "Z"                                Z"
120 c$(3)= "Z"                                Z"
125 c$(4)= "Z"                                Z"
130 c$(5)= "Z  ZZZ  ZZZZ  ZZZ  Z"              Z"
140 c$(6)= "Z  ZZZ  ZZZZ  ZZZ  Z"              Z"
150 c$(7)= "Z  ZZZ  ZZZZ  ZZZ  Z"              Z"
```

```

160 c$(8)="Z" Z"
170 c$(9)="Z" Z"
190 c$(10)="Z" Z"
195 c$(11)="Z ZZZZZ ZZZZZZZZ Z" Z
200 c$(12)="Z ZZZZZ ZZZZZZZZ Z" Z
210 c$(13)="Z" Z"
220 c$(14)="Z" Z"
230 c$(15)="Z" Z"
240 c$(16)="ZZZZZZZZZZZZZZZ ZZZZ" ZZZZZZZZZZZZZZ
250 c$(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)="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ.
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
300 gosub 550: ' prints the layout
310 xa=40: xn=xn: ya=150: yn=ya: ' coordinates of the character
311 |SETUPSP,0,0,&x111: ' collision detection with sprites and layout
312 |SETUPSP,0,7,1: ' sequence = 1
320 |LOCATESP,0,ya,xa: 'place the character (without printing it)
325 cl%=0:'declare collision variable, explicitly integer (%)'

330 '----- game cycle -----
340 gosub 1500:'keyboard reading and character movement routine
350 |PRINTSPALL,0,0
360 goto 340

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

1500 ' character movement routine -----
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,@cl%,0:IF cl%=0 THEN 1536
1534 yn=ya:|POKE, 27001,yn:|COLAY,@cl%,0:IF cl%=0 THEN 1536
1535 xn=xa: yn=ynn:|POKE, 27001,yn:|POKE, 27003,xn:|COLAY,@cl%,0:IF cl%=1 THEN
yn=ya:|POKE,27001,yn
1536 ya=yn:xa=xn

```

**1537 RETURN**

### 10.3 How to open a gate in the layout

If you want your character to be able to take a key and open a gate or generally remove a part of the layout to allow access, you have to do two steps:

- 1) Have an 8x8 wipe sprite defined where everything is zeros. Using |LAYOUT you print it in the positions you want
- 2) Then, using |LAYOUT again, you print spaces where you have deleted. Then the layout map will be left with the " " character in those positions and the collision function with the layout will result in zero.

In the game "Mutant Montoya" this technique is used to open the castle gate, as well as to open the gates leading to the princess.



Fig. 62 Layout modification when the key is picked up

The following example illustrates the concept by opening a gate at coordinates (10, 12) with a size of 2 blocks, by picking up a key that is defined with sprite 16.

As soon as you pick up the key, the gate opens and the key is deactivated in order not to evaluate the collision with it any more, that is to say, the |COLSP command will return a 32 from the moment you pick up the key if you collide with it again.

After opening the gate, if you move the character to the place where the gate used to be, the collision with the layout will result in 0.

```
'----- this part is inside the logic loop ----- 6410
|PRINTSPALL,1,0
6411 |COLSP,@cs%,0:IF cs%<32 THEN IF cs%>=15 then gosub 6500 ( . .
. . more instructions . . . )

'----- gate opening routine -----
6499 ' check that your collision is with the key, which is sprite 16 6500
borra$="MM":spaces$=""      ':' the deletion sprite has been defined as
" M" (M is sprite 18 in the "language" of the |LAYOUT command)
6501 if cs%=16 then |LAYOUT,10,12,@delete$ : |LAYOUT,10,12,@spaces$:
|SETUPSP,16,0,0,0
6502 return
```

## 10.4 A puzzle game: LAYOUT with a background

Next, we are going to see an example that uses layout and overwriting, for which it establishes an ASCII threshold that allows the |COLAY command not to consider collision with the background elements. Specifically, the background element is the letter "Y", which corresponds to the sprite id= 30, and the ASCII of the "Y" is 89.



Fig. 63 Layout with a background pattern and overwriting

As you can see in the example it is only necessary to invoke COLAY with the threshold parameter once, since the successive invocations already take this threshold into account.

Another interesting aspect is the keyboard management of this example. It is optimal for executing the fewest number of operations |COLAY and at the same time it gives a very pleasant feeling when moving down a corridor and connecting to another corridor with two keys pressed at the same time.

```
10 MEMORY 23999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
21 on break gosub 5000
25 call &bc02:'restore default palette just in case'.
26 gosub 2300:' palette with overwriting
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,0,80,0,200: ' limits of the game screen
45 |SETUPSP,30,9,&84d0:' background grid ("Y")
46 |SETUPSP,31,9,&84f2:' brick ("Z")
50 dim c$(25):for i=0 to 24:c$(i)=" ":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
110 c$(2)= "ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
120 c$(3)= "ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
125 c$(4)= "ZYZZZYZZZZZZZZZZZZZZZZZZZZZZZZZZYYZ".
130 c$(5)= "ZYZZZYZZZZZZZZZZZZZZZZZZZZZZZZYYZ".
140 c$(6)= "ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
150 c$(7)= "ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
160 c$(8)= "ZYZZZZZZZZZZZZZZZZZZZZZZZZZZZZYYZ".
170 c$(9)= "ZYZ      ZYZ      ZYZ"
190 c$(10)="ZYZ      ZYZ      ZYZ"
195 c$(11)="ZYZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
200 c$(12)="ZYYYYYYYYYYYYYYYYYYYYYYYYY
              YYYYYYYZ".
210 c$(13)="ZYYYYYYYYYYYYYYYYYYYYY
              YYYYYYYZ".
220 c$(14)="ZYZZZYZZZZZZZZZZZZZZZZZZZZZZ".
230   c$(15)="ZYYYYYYYYYYYYYYYZ
              YYYYYYYYYYYYYYYZ"
```

```
240      c$(16)="YYYYYYYYYYYYYYYYZ  
              ZYYYYYYYYYYYYYYZ"  
250  c$(17)="ZYZZZZZZZZYZZZZZZZZZ  
              ZZZZZZZYZ".  
260  c$(18)="ZYYYYYYYYYYYYYYYYYYYYYYYY  
              YYYYYYYZ".  
270  c$(19)="ZYYYYYYYYYYYYYYYYYYYYYYYY  
              YYYYYYYZ".
```



## 10.5 How to save memory in your layouts

If your game has many screens and you need to save space you can use many simple techniques to save memory. One screen consumes almost 0.5 KB so it is important to use methods to reduce its size

The easiest way to do this is to edit the screens as if they were sprites. You edit them with SPEDIT (for example) and each pixel will represent an element of the layout. Depending on the colour, it will represent rocks, bricks, empty space, water, earth, etc. As there are 16 colours in mode 0 available, you will have 16 types of bricks. The sprite you generate you will have to store it as an image and before displaying it on screen, convert it to layout, scanning it pixel by pixel and transforming it into the ASCII code you need (you will have to program it in BASIC or whatever you want). If we consider that you will use 5 lines of characters for the game markers, one screen will consume 20x20 pixels = 400 pixels = 200 Bytes. Therefore, in 1KB you can fit 5 screens and in 10KB you can fit 50 screens. You will have used 4 bits per layout element.

Editing screens as if they were sprites is very "visual", although you can decide to use fewer bits per layout element. If you use only 2 bits, you will have 4 types of elements and you can also edit screens as if they were images, using MODE 1. In that case you will be able to fit 100 screens in 10KB. If you use a different number of bits per brick it gets complicated because you can't draw the screens as if they were sprites, but you can program something that converts an image you draw into the bits you need.

Another simple and effective solution is to define each layout with large blocks, e.g. 8x16 pixels. And build the screens by defining them with characters that we can store in memory. In the video game "**Happy Monty**", the first screen of "Mutant Monty" is recreated with a matrix of 16x10 characters = 160 bytes, so that 25 screens occupy 4 KB. It is true that this layout is only 16 blocks wide and not the 20 that it can be, and also only 10 blocks are defined vertically, instead of the 25 that it can be, but this way it occupies very little memory and we can create many screens.

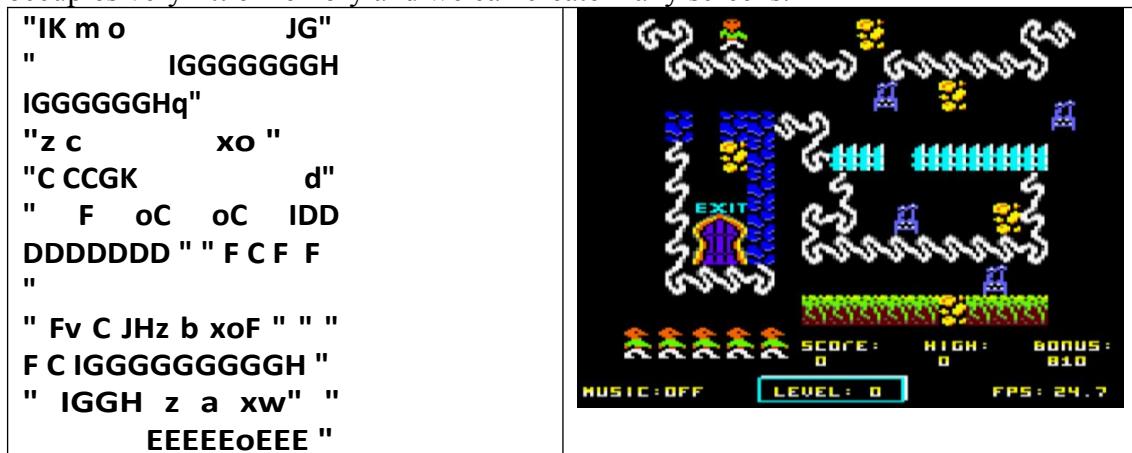


Fig. 64 a layout defined with 160 characters



# 11 Advanced programming and "bulk logics".

## 11.1 Measurement of the speed of commands

The BASIC interpreter is very heavy in execution because it not only executes each command, but also analyses the line number, parses the entered command, validates its existence, the number and type of parameters, that its values are in valid ranges (for example, PEN 40 is illegal) and many other things. It is the syntactic and semantic analysis of each command that really weighs and not so much its execution. The case of RSX commands is no exception. The BASIC interpreter checks their syntax and that weighs a lot, even though they are routines written in ASM, because before invoking them, the BASIC interpreter has already done many things.

Therefore, you have to save command executions by cleverly programming so that the logic of the program goes through as few instructions as possible, even if this sometimes means writing more. An indispensable practice is to use instructions that move or affect a group of sprites, such as COLSPALL,

|AUTOALL or |MOVEALL, avoiding the use of loops with instructions that affect a single sprite.

A decisive factor when invoking a command is the number of parameters. The more parameters it has, the more expensive it is for BASIC to interpret, even if it is an ASM routine that is invoked by CALL, because the CALL command is still BASIC and before accessing the routine in ASM, the number and type of parameters are inevitably analysed.

To evaluate the cost of executing a command you can use the following program. You can also use it to evaluate the performance of new assembler functions that you incorporate into the 8BP library if you wish to do so.

```
1 call &6b78
10 MEMORY 23499
11 DEFINT a-z
12 c%=0: a=2
30 FOR i=0 TO 31:|SETUPSP,i,0,0,0:NEXT:'reset
31 iterations=1000
40 a!= TIME
50 FOR i=1 TO iterations
60 <here you put a command, e.g. PRINT "A">
70 NEXT
80 b!=TIME
90 PRINT (b!-a!): rem what it takes in cpc. units of time (1/300 seconds)
100 c!=((b!-a!)*1/300)/iterations: rem c! = how long each iteration takes
in seconds
120 d!=(1/50)/c!
130 PRINT "you can execute ",d!, "commands per sweep (1/50 sec)".
140 PRINT "command takes ";(c!*1000 -0.47); "milliseconds"; "command takes
";(c!*1000 -0.47); "milliseconds".
```

Note: For assembly language experts, you should note that if you intend to measure the execution time of a routine that internally disables interrupts (uses DI, EI instructions), the time elapsed during disabling is not measurable with this BASIC program. The 8BP commands do not disable interrupts and are all measurable.

Let's see below the performance result of some commands (measured with the above program). It must be said that it is faster to execute a direct call to the memory address (a CALL &XXXX) than to invoke the corresponding RSX command. In the following table obviously the lower the result (expressed in milliseconds), the faster the command. The table presented here should be kept in mind at all times and you should make your programming decisions based on it. It is a table with measurements of BASIC commands and 8BP commands.

Command	ms	Comment
<b>PRINT "A"</b>	3.63	Very slow. Don't even think of using it, except occasionally to change the number of lives, but don't print score in a game for every enemy you kill.
<b>LOCATE 1,1: PRINT points</b>	24.8 + 7	Placing the text cursor with LOCATE and printing the moon value variable "points" is very expensive. If you update points, do it only from time to time and not every game cycle.
<b>C\$=str\$(points)  PRINTAT,0, y, x, @c\$</b>	10	Printing the dots using PRINTAT is much more efficient than using LOCATE + PRINT (=32 ms) , but still expensive. Use PRINTAT sparingly.
<b>REM hello</b>	0.20	Comments consume
<b>' hello</b>	0.25	You save 2 bytes of memory, but it's slower!
<b>GOTO 60</b>	0.19	Very fast! Even faster than REM. Use this command mercilessly, use it!!!
<b>A = 3</b>	0.55	A simple assignment costs. Everything costs, every instruction must be thought through.
<b>A = B</b>	0.72	Assigning the value of one variable to another is more expensive than assigning a value. And assigning the value of an array is even more expensive, because accessing the array costs. And if the array is two-dimensional, it costs even more.
<b>A = miarray(x)</b>	1.33	
<b>A= miarray(x,y)</b>	1.84	
<b> LOCATESP,i,10,20</b>	2.8	If you do not use negative coordinates it is better to use the BASIC POKE command to set coordinates.
<b> LOCATESP,i,y,x</b>	3.22	If the coordinates are variable then it takes longer.
<b>CALL &amp;XXXX,i,x,y</b>	1.81	The CALL equivalent is much faster.
<b> MOVER,31,1,1</b>	3.23	It is somewhat slow and should therefore be used sparingly.
<b>CALL &amp;XXXX,31,1,1</b>	1.77	The equivalent call is much faster
<b>POKE &amp;XXXX, value</b>	0.71	Very fast! Use it to update sprite coordinates (if positive). POKE does not accept negative numbers, but you can use the formula 255+x+1 if you want to enter a negative number. For example, to enter a -4 you would enter 255-4+1=252 Another simple way to enter positives and negatives is to use POKE address, x and 255.
<b>POKE dir,data</b>	0.85	Very fast considering that it must also translate the variable "dir".
<b> POKE,&amp;xxxx,value</b>	2.5	It allows negative numbers and if you only update one coordinate (X or Y) it is better than LOCATESP.

<b>X=PEEK(&amp;xxxx)</b>	0.93	Very fast! Depending on the type of game, it can be an alternative to COLSP, by looking at the colour of a screen memory address. In the appendix on video memory I explain how to do this.
<b>X=INKEY(27)</b>	1.12	Very fast. Suitable for video games, although you must use it intelligently as recommended in this book.
<b>IF x&gt;50 THEN x=0</b>	1.42	Each FI weighs, we must try to save them because a game logic is going to have many
<b>IF A=value THEN GOTO 100</b>	1.24	Both sentences are equivalent but the second one takes less time.
<b>Vs</b>	Vs	
<b>IF A=value THEN 100</b>	1.18	
<b>IF inkey(27)=0 then x=5</b>	1.75	Acceptable. It's faster than doing b=INKEY(27) and then the IF...THEN
<b>10 IF inkey(27) then 30 20 x=5 30 &lt;instructions&gt;.</b>	1.0	A much more efficient way to do the same
<b>IF x&gt;0 then</b>	1.3	In BASIC it is possible to save 0.5ms taking into account that any non-zero value means TRUE.
<b>Vs</b>	Vs	If we want to control a specific value we will do:
<b>IF x then</b>	0.8	<b>10 IF x&gt;20 THEN 30 20 &lt;things to do if x=20&gt; 30</b> ...
		The use of this technique is highly recommended in keyboard reading.
<b>A=A+1: IF A&gt;4 then A=0</b>	2.6	It is much better to use the second option (using MOD). On the other hand, the use of MOD should be done with caution. If we do: <b>A=(A+1) MOD 3</b>
<b>Vs</b>	Vs	
<b>A=A MOD 3 +1</b>	1.7	It costs us 2 ms because the brackets are very expensive and yet we get the same thing. If we want to count from a number other than 1, we put in any odd number and that will do it.
<b>A=A MOD 3 + &lt;impar&gt;</b>		There is an even quicker way to do this, with the binary AND operator, which we will now look at.
<b>A=1+A AND 7 (initial value 0 Final value 7)</b>	1.6	<b>This allows you to vary a variable cyclically between N values</b> so that you can choose a sprite ID for your new shot or for an enemy that enters the screen.
<b>A=20 +A MOD 7 (initial value 0 Final value 26)</b>	1.88	It is better to use AND than MOD, because AND is a fast binary operation and MOD involves division, which is very expensive for our beloved Z80 microprocessor. However, if we need to use ID sprites that do not start with 1, then we will need parentheses because the "+" operator <b>has priority over "AND"</b> and the speed advantage of AND is lost. In that case
<b>A=21 + (A and 7) (initial value 21 Final value 28)</b>	1.95	

		is better MOD. Anyway, always try it and choose because depending on the initial number to add you will get different times. Unbelievable but true 20+A mod 7 → takes 1.88 29+A mod 7 → takes 1.94
<b>If A &lt;0 then A=15 A=A AND 15</b>	1.71 1.24	<b>Check that a number is not negative</b> You can simplify the check because a negative number is actually a number that has a "1" in the most significant bit, and we remove the negativity with a simple and
:	0.05	It doesn't save much, but it is faster to use ":" instead of a new line number, and if you apply this many times you end up saving significantly. Two instructions on two lines will spend 0.03ms more than if both are on the same line separated by ":".
<b> PRINTSP,0,10,10</b>	5.3	Single 14 x 24 sprite (7 bytes x 24 lines) If you are going to print several sprites, it is much better to print all the sprites at once with PRINTSPALL.
<b>CALL &amp;xxxx,0,10,10</b>	3.5	Equivalent to PRINTSP, so faster, but less readable.
<b> PRINTSPALL (32 sprites 8x16 of mode 0, i.e. 4 bytes x 16 lines)</b>	55.4	That's about 18 fps at full sprite load. What it takes is $T = 3.25 + N \times 1.7$ That is, 1.7 ms per sprite and a fixed cost of 3 ms. This fixed cost is the cost of BASIC parsing plus the cost of going through the sprite table looking for sprites to print. If you omit the parameters (it is possible and you would take the values of the last invocation), you save 0.6ms in the fixed part, that is: $T = 2.6 + N \times 1.1$ If the printing is overprinted and/or flip-printed, it is more expensive. The relative costs of each type of printing are shown below: <b>Normal printing: 100%.</b> <b>Print with overwrite: 164%</b> Print flipped: <b>179%</b> Print with overwrite: 164% Print flipped: 179 Printing flipped with overwriting: 220%.
<b> PRINTSPALL,N,0,0 (no sprite active)</b>  N=0 N=10 N=31	2.6 4.3 5.9	Cost of ordering sprites: When N=0, with no sprites to print, the function has to run through the sprite table sequentially. But going through it in order is more expensive, as evidenced by the time consumed as N increases. The time difference (5.9 - 2.6 = 2.5ms) is what it costs to sort all the sprites.
<b> COLAY,@x%,0</b>	3.0 Vs	Use only on the character, not on enemies or the game will slow down. If the character is multiples of 8 it is faster. In this example it was 14x24 and logically 14x24 was the size of the character.

<b> COLAY</b>	2.4	is not a multiple of 8. the larger the sprite the longer it takes. If you invoke the command without parameters it is much faster (you save 0.6 ms)!
<b> COLAY vs CALL &amp;XXXX</b>	2.4 vs 2.0	Using CALL as usual is faster, but less readable.
<b>GOSUB / RETURN</b>	0.56	Acceptably fast. The measurement was done with a routine that only does return.
<b> SETUPSP, id, param, value</b>	2.7	Acceptable, although POKE is much better for certain parameters. There are parameters that can be set with POKE such as status, but not others (such as a route). See the reference guide
<b>FOR / NEXT</b>	0.6	You can use it to run through several enemies and have each one move according to the same rule. You should consider whether you can use AUTOALL or MOVEALL for your purposes as one command will move everyone you want, which is much better than a loop.
<b> COLSP,31, @c%</b>	5.5	It takes about the same regardless of the number of active sprites. Avoid always calling with the collision variable to speed it up to 4.3 ms.
<b> COLSP,31</b>	4.3	If you have one ship or character and several shots, it is much more efficient to invoke COLSPALL instead of invoking COLSP several times.
<b> ANIMALL (this command is only available with a parameter from PRINTSPALL, it is not available from can be invoked directly)</b>	3.5	It is expensive but there is a way to invoke it in conjunction with invoking  PRINTSPALL , via a parameter that causes this function to be invoked before printing the sprites. This saves the BASIC layer, i.e. the time it takes to send the command, which is >1ms. Therefore we can say that this command will normally consume a little less than 2ms.
<b> AUTOALL</b>	2.76	It is inexpensive and can move all 32 sprites at once.
<b> MOVERALL,1,1</b>	3.4	It is not very expensive and can move all 32 sprites at the same time.
<b>SOUND</b>	10	The sound command is "blocking" as soon as the 5 note buffer is full. This means that your BASIC logic must not chain more than 5 SOUND commands or it will stop until some note finishes...If you decide to use it, you must be very careful because it is very consuming. execution time (10 ms is very long)
<b>IF a&gt;1 AND a&gt;2 THEN a=2</b>	2.52	A simple way to save 0.13 ms
<b>Versus</b>	Vs	In everything you program, keep these details in mind, every saving is important.
<b>IF a&gt;1 THEN IF a&gt;2 THEN a=2</b>	2.39	

<b>A=RND*10</b>	4.2	The BASIC RND function is very expensive. You can use it, but not in every game cycle, but only eventually, for example, when a new enemy or the like. Another simple solution is to store
		10 random numbers in an array and use them instead of invoking RND
<b>Border &lt;x&gt;</b>	0.75	Quite fast. Useful to use in combination with some kind of sprite collision, reinforcing the explosive effect.
<b>IF a AND 7 then 30</b>	1.19	I have put the execution time when the condition is met.
<b>IF A MOD 8 then 30</b>	1.29	Both cases are quite fast.
<b>ON x GOTO L1,L2,L3,L4</b>  Vs  <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	An ON GOTO command is on average 1 ms faster than its equivalent with "IF" commands, although it also depends on the probability of occurrence of each of the 4 values. If the probability of the 4 values is the same, we can save 1ms using ON GOTO It can be further optimised as follows  <b>10ON X GOTO 30,40,50</b> <b>20 &lt;instructions case x=4&gt;: GOTO 60</b> <b>30 &lt;instructions case x=1&gt;: GOTO 60</b> <b>40 &lt;instructions case x=2&gt;: GOTO 60</b> <b>50 &lt;instructions case x=3&gt;: GOTO 60</b> <b>60 continuation of the programme</b>  With this strategy we are down to 3.54 ms.

*Table 5 List of execution times of some instructions*

### Important recommendations:

- **Use DEFINT A-Z at the beginning of the programme.** The performance will improve a lot. This is almost mandatory. This command deletes any variables that existed before and forces all new variables to be integers unless otherwise indicated by modifiers such as "\$" or "!" (See the Amstrad BASIC programmer's reference guide). Note that when using DEFINT, if you want to associate a number greater than 32768 you will have to do it in hexadecimal.
- If you can avoid going through an IF by inserting a GOTO, it will always be preferable to
- When you lack speed and need a little more speed, use **CALL. <address>** instead of RSX. If you do this, you must pass parameters containing negative numbers in hexadecimal format.

- Do not synchronise the **|PRINTSPALL** command with the screen sweep unless your game is running very fast. Synchronising can reduce your FPS. In general, as long as you get 12 FPS your game will be "playable".

- **Eliminate whitespace.** Each blank space in your BASIC listing consumes 0.01ms in execution.
- **Shorten the name of your variables.** The longer they are, the more they cost to access.

operation	Weather
A=A+1	One letter, takes 1.18ms
HO=HO+1	Two letters, takes 1.2ms (2% longer)
HELLO=HELLOA+1	5 letters, takes 1.25 ms (6% longer)
HELLOFRIENDS=HELLOFRIENDS +1	10 letters 1.34 ms (13% more)

- **Reduce the number of variables.** If there are many variables, read and write accesses are slower.
- Once you have invoked with parameters the **|STARS** command or the **|PRINTSPALL**, or **|COLAY** or other 8BP commands, the following times do not invoke it with parameters. The 8BP library has "memory" and will use the last parameters you used. This saves milliseconds by going through the parsing layer of the BASIC interpreter.
- Always keep in mind that a non-zero expression is TRUE. This will allow you to save 0.5ms on each IF and can be used in keyboard reading and variable control.

Bad option	Good choice (saves 0.5ms)
<b>IF x&lt;&gt;&gt;0 THEN &lt;instructions&gt;</b>	<b>IF x THEN &lt;instructions&gt;</b>
<b>IF x=20 THEN...</b>	<b>10 IF x-20 THEN 30 20 &lt;instructions&gt;. 30</b>
<b>IF INKEY(34)=0 THEN &lt;instructions&gt;.</b>	<b>10 IF INKEY(34) THEN 30 20 &lt;instructions&gt;. 30</b>

- In ship games where you don't use overwrite, make sure your ship is sprite 31, so that it will go "over" the sprites that pretend to be the background, as your ship will be printed afterwards.
- Test alternative versions of the same operation **A=A+1:IF A>4 then A=0 : REM this consumes 2.6ms** **A=A MOD 3 +1 : REM this consumes 1.84 ms** **A=1 + A AND 3 : REM this consumes 1.6 ms**
- Avoid using negative coordinates. This will allow you to use POKE to update your character's position. The POKE command (the BASIC one) is very fast but only supports positive numbers, as does PEEK. In case you use native coordinates, use **|POKE** and **|PEEK** (8BP commands). Reserve the use of **|LOCATESP** for when you are going to modify both coordinates at the same time and they can be positive and/or negative. Remember also that a POKE of a negative x-value can be done using **POKE address, 255+x+1**. In case you want to use negative coordinates to show how the enemies are slowly entering the screen from the left side (clipping), you can avoid the coordinates

The use of a **|SETLIMITS** and thus produce the same effect with coordinates starting at zero and a slightly smaller game screen.

- If you need to check something, don't check it every game cycle. It may be enough to check that "something" every 2 or 3 cycles, without needing to check it every cycle. To be able to choose when to execute something, make use of "modular arithmetic". In BASIC you have the MOD instruction which is an excellent tool. For example, to execute one out of 5 times you can do: **IF cycle MOD 5 = 0 THEN ...** although it is better to use **AND** operations than **MOD** operations.
- Make use of "**death sequences**". This will allow you to save instructions to check if an exploding sprite has reached its last animation frame in order to deactivate it.
- Overwriting is expensive: if you can make your game without overwriting you will save milliseconds and gain colour. Use it when you need it, but not for no reason.
- Animation macros save you lines of BASIC because you don't need to check the direction of movement of the sprite. Use them whenever you can.

**11.2** Our Amstrad only has 64KB and if you discount the video memory, the memory for your sprites, your music and the 8BP library, you are left with 24KB of BASIC to use.

very well. If each screen has its own "programme" within your game, you will barely be able to make a set of 10 screens.

There are two things you should try to do to reduce memory usage

- Create low-byte screens
- Create a single logic that governs all screens, i.e. the same game cycle is to be executed on all screens in the game.

In screen-passing games that use layout, each screen can occupy 20x25 bytes, i.e. 500 bytes. If you use certain "tricks" as explained in chapter 8, you can reduce this memory. In the video game "**Happy Monty**", 25 screens are built with only 160 bytes each and there is a single game cycle logic for all screens.

**It is very important that you program a single game cycle logic and apply it to all screens.** If you program a game cycle logic for each screen, the source code of your program will be huge and you will be able to program few screens because you will run out of memory soon.

You can also overcome memory limitations by using algorithms that generate mazes, or screens without storing them. This way you can make many more screens. This requires creativity, of course, but it is possible. An algorithm

always takes up less space than the data it generates, although logically it takes more time to run than simply reading stored data.

You can reuse enemy logic from one screen to another, saving lines of code. Take advantage of the GOSUB/RETURN mechanism for this. It is also very useful to use routes on enemies. **With the pathing mechanism, the enemy moves without executing BASIC logic** and will work very fast. Just assign a path to an enemy and it will run it over and over again without the need for expensive IF statements, assignments, etc.

You can also make games that load in stages, so you don't have the whole game in memory at once. This is a bit annoying for the tape user (CPC464) but not for the disk user (CPC6128).

Use **sprite flipping** to save memory on your sprites

### **11.3 Mass Logic" technique**

You will often need to move a lot of sprites, especially in space arcade or "commando" style games (Capcom's 1985 classic).

You could act separately on the coordinates of all the sprites and update them using POKE, but it would be very slow, unfeasible if you want fluidity of movement. The best (and easiest) way is to make combined use of the automatic movement and relative movement functions, which are |AUTOALL and |OVERALL respectively.

The key to achieving speed in many sprites is to use the technique I have dubbed "massive logic". This technique is essentially about running less logic per game cycle (called "reducing computational complexity") and there are several options for doing this:

- Use **a single logic** that affects many sprites at once (using the automatic and/or relative movement flags).
- Execute several tasks, but only one or a few of them in each game cycle, using **modular arithmetic (or binary operations) in cascade**.
- Introduce **limitations in the game that are not important** or do not affect gameplay, to reduce the number of tasks that are executed in each game cycle or to simplify tasks so that they are executed faster.
- As a general rule, reduce the number of instructions your program goes through in each game cycle, sometimes replacing algorithms with pre-calculations or putting in more instructions so that (paradoxically) fewer are executed each cycle.

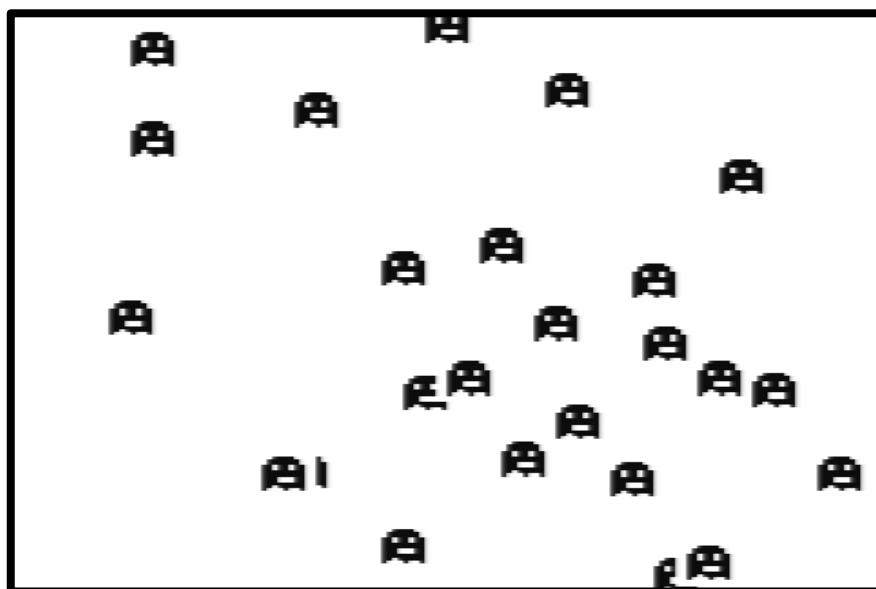
These ideas have the same goal: to **execute less logic in each cycle**, allowing all sprites to move at the same time, but making fewer decisions in each cycle of the game. This is called "**reducing computational complexity**", **transforming a problem of order N (N sprites) into a problem of order 1 (a single logic to be executed in each frame)**.

The key is to determine which logic or logics to run in each cycle. In the simplest case, if we have N sprites we will simply run one of the N logics. But in more complex cases we will have to be astute in determining which logics to run.

### 11.3.1 Moves 32 sprites with massive logics

Now let's look at a simple example of moving 32 sprites simultaneously and smoothly (at 14fps). It is perfectly possible. Only one ghost will make decisions in each cycle, although all ghosts will move in all cycles. We can also animate them all (associating them with an animation sequence and using

|PRINTSPALL,1,0 ) and it will still be smooth, but it will still look like there is more movement as the flapping of a fly's wings (for example) generates a lot of feeling of movement.



*Fig. 65 with massive logics you can move 32 sprites simultaneously*

What we have done is to reduce the computational complexity. We started with an "order N" problem, where N is the number of sprites. Assuming that each sprite logic requires 3 BASIC instructions, in principle,  $N \times 3$  instructions would have to be executed in each cycle. With the "bulk logic" technique, we transform the "order N" problem into an "order 1" problem. An "order 1" problem is one that involves a constant number of operations regardless of the size of the problem. In this case we have gone from  $N \times 3 = 32 \times 3 = 96$  BASIC operations to only 3 BASIC operations. This reduction in complexity is the key to the high performance of the bulk logic technique.

```

1 MODE 0
10 MEMORY 24999: CALL &6B78
20 DEFINT a-z
25 ' reset enemies
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
35 ' num enemies 12 x 16 (6bytes wide x 16 lines)
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 chance=rnd*3
170 if random=0 then |SETUPSP,i,5,2:
|SETUPSP,i,6,0:return
180 if random=1 then |SETUPSP,i,5,-
2:|SETUPSP,i,6,0:return
190 if random=2 then
|SETUPSP,i,5,0:|SETUPSP,i,6,1:return
200 if random=3 then |SETUPSP,i,5,0:|SETUPSP,i,6,-1:re

```

### 4.3.2 Alternating and periodic cascade execution

You do not need to perform all tasks in every game cycle. For example, if you want to check if the shot has left the screen, you can check every second or third cycle, instead of checking every cycle.

You can also make the enemies shoot every few cycles and not every cycle (otherwise they would shoot you a lot!!).

In short, there are things you don't need to do in every cycle and you can save instruction execution per cycle and therefore you will achieve higher speed. This is actually the basic foundation of the "bulk logic" technique.

There are two basic techniques for this: The use of modular arithmetic and binary **AND** operations. Binary **AND** operations are faster than **MOD** operations.

Technique	Time consumed
<b>A = A+1: if A =5 then A=0: GOSUB &lt;routine&gt;.</b>	2.6 ms
<b>IF cycle MOD 5 =0 THEN gosub routine</b>	1.84ms, assuming you already have a variable called cycle that is updated

The MOD operation is somewhat costly and therefore a binary operation is sometimes better.

Assuming you have the cycle variable that is updated every time, we can do a binary operation to see when a group of bits gives a certain value. For example, if we look at the 4 least significant bits of the cycle variable, they will always go from 0000 to 1111 and back again. Well, if we do an AND 15 with that variable we can do the same as with MOD 15. The number 15 in binary is 1111 and so an AND reveals the value of those 4 bits.

Technique	Time consumed
-----------	---------------

If cycle AND 15=0 then gosub routine	1.6 ms (executes one in 16 times)
If cycle AND 1=0 then gosub routine	1.6ms (Runs once every 2 times)

If you have several periodic things to run you can do it like this:

```
c=cycle AND 15 :' rem 15 is in binary 1111
IF c=0 THEN GOSUB <routine1> (routine1 is executed once every 16 times) iF
c=8 THEN GOSUB <routine2>... ( routine2 is executed once every 16 times,
but far away in time from the execution of routine1)
```

In this way you are spreading the time over different tasks, so that in each cycle you only do one task, but after several cycles you have done all the tasks.

In order to check the cycle variable and decide to execute a task, there is a better way to execute the binary operations and it is the following:

Technique	Time consumed
<b>10 If cycle and 7 then 30</b> <b>20 &lt;instructions which are executed every 8 cycles&gt;</b> <b>30 &lt;programme continuation&gt;</b>	<b>1.18 ms.</b> This is undoubtedly the best strategy for periodic task execution.

Applying the same strategy to MOD also increases the speed, although less than with AND. However, it is very good because it works, even if the period is not a multiple of 2 (you can put MOD 7, MOD 5, MOD 10, etc.).

Technique	Time consumed
<b>10 If cycle MOD 8 then 30</b> <b>20 &lt;instructions which are executed every 8 cycles&gt;</b> <b>30 &lt;programme continuation&gt;</b>	<b>1.29 ms.</b> Almost as good as AND and the best strategy when we need a period that is not a multiple of 2.

As you can see, for each task we want to introduce in the logic, we must introduce an "IF". **However, it can still be improved** and made more efficient by using task execution time intervals that are multiples. If they are multiples, **the "IF" of task 2 can be done within task 1, and the "IF" of task 3 within task 2, in "cascade"**. This greatly reduces the number of IFs we execute in each cycle, being in many cases a single IF.

Let's look at a complete example, which allows you to multitask 4 different tasks, but reducing the number of tasks running at the same time. The following sequence represents the order of execution of the tasks and then the source code.



```
10 IF cycle AND 1 THEN 90
20 REM every two cycles we enter here
25 IF cycle AND 3 THEN 80
30 REM every 4 cycles we enter here
35 IF cycle AND 7 THEN 70
40 REM every 8 cycles we enter here
50 <task 4> : GOTO 100
70 <task 3> : GOTO 100
```

```

80 <task 2> : GOTO 100
90 <task 1>.
100 REM --- end of tasks ---

```

In this example we have chosen the intervals 2, 4 and 8.  
 AND 1 : this gives me an interval of 2 because it is zero every 2 cycles  
 AND 3 : it is zero every 4 cycles  
 AND 7 : is zero every 8 cycles

Because we have chosen operations in multiple intervals, the IFs are executed "in cascade": we only enter an IF if we have entered the previous one:

- Half of the cycles execute a single IF (line 10).
- Half of the cycles execute two IF statements (lines 10 and 25), of which half (i.e. 25%) will execute three IF statements (lines 10, 25 and 35).

On average  $1*50\%+2*25\%+3*25\% = 1.75$  IF statements are executed per cycle.

Thanks to this strategy of **using modular arithmetic with binary operations in multiple intervals to cascade them**, we can reduce the number of "IF" operations to a minimum and at the same time reduce the computational complexity from order N (n tasks) to order 1 (one task per cycle). This speeds up your games tremendously.

### 11.3.3 Simple example of mass logic

In the video game "Mutante Montoya", the enemy sprites take turns to run through the different cycles of the game. **When I programmed this game, I had not yet programmed the |ROUTEALL mechanism that allows to assign fixed trajectories to the sprites, but I was able to solve it with massive logics.** In case you wanted to make a game where the enemies have "intelligence", a fixed path would not work, so even if you have the ROUTEALL command, you would have to rotate the sprite logic as described below, so this example is interesting.

Suppose we have 3 enemy soldiers moving from right to left and left to right. To gain speed we will run only one soldier's logic in each game cycle.

In order to keep the x-coordinate of each soldier moving forward despite this, we will use the automatic movement flag instead of updating it ourselves.

```

10 MEMORY 24999
20 MODE O: DEFINT A-Z: CALL &6B78:' install RSX
25 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
26 |SETLIMITS,0,80,0,0,200
30 'parameterisation of 3 soldiers
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: 'we place the sprites
81 i=0
89 ----- MAIN GAME LOOP (GAME CYCLE) -----
90 i=i+1:gosub 100
92 if i=3 then i=0
93 |AUTOALL
94 |PRINTSPALL,1,0: ' animates and prints the 3 soldiers
95 goto 90
96 ----- END-OF-GAME-CYCLE -----
99 '---- soldier routine ----
100 |PEEK,27003+i*16,@x%: x(i)=x%
101 IF address(i)=0 THEN IF x(i)>=xmax(i) THEN
address(i)=1:|SETUPSP,i,7,10: |SETUPSP,i,6,-1 ELSE return
110 IF x(i)<=xmin(i) THEN address(i)=0:|SETUPSP,i,7,9 :
|SETUPSP,i,6,1
120 return

```

Each soldier has his own logic, but we only execute one in each game cycle, making the game cycle much lighter.

The only limitation is that by running each soldier's logic one out of 3 times, the coordinate could exceed the limit we have set for two cycles. That makes us need to be more careful when setting the limit, making sure when we run it that it never invades and erases a wall of our screen maze, for example. I will try to explain this problem more precisely:

Let's say we have 8 sprites and our sprite moves in every cycle, but we only execute its logic one out of 8 times. Imagine a sprite that is at position x=20 and we want it to move to position x=30 and turn around. Let's consider that the sprite has an automatic movement with Vx=1. In that case we will check its position when x=20, x=28, x=36. When we reach 36 we will realise that we have gone too far!!! and we will change the sprite's velocity to Vx=-1.

As you can see, the control of the trajectory limits is not precise, unless we take this circumstance into account and set the limit to something we can control, which will be Xfinal = Xinitial + n\*8.

This limitation is minuscule compared to the advantage of moving many sprites at high speed. With some cunning we can even run the logic fewer times, so that only every second cycle some kind of enemy logic is executed.

#### 11.3.4 Block" movement of squadrons

If you simply want to move a squadron in one direction at a time, either of the following two functions from the 8BP library will work:

- If you use **|AUTOALL** you have to auto-speed the sprites in the direction you want (in Vx, in Vy or both) and of course set bit 4 of the status byte. The AUTOALL command has an optional parameter to internally invoke |ROUTEALL before moving the sprites.

- If you use **|MOVERALL** you have to set bit 5 of the status byte to the sprites you are going to move. This command requires as parameters how much relative movement in Y and X you want.

In this way, with a single instruction you are moving many sprites at the same time. In case each of your sprites must move independently and with an independent logic, as it happens in games like "pacman", you will have to be more cunning, as I will tell you next.

### **11.3.5      Massive logic technique in "pacman"-type games**

If you have many enemies and they must make decisions at every fork in a maze, it is not a good strategy to simply take turns running enemy logic every cycle. What is best is to execute logic when that logic must make a decision. In pacman-type games this happens when a ghost reaches a fork where it can take a new direction of movement based on its intelligence. This can be done with a simple "trick". It is simply to place enemies in well-chosen positions at the start of the game.

Suppose you have 4 enemies and the forks of the maze occur in multiples of 4. If the first enemy is in a position multiple of 4, it is his turn to execute his logic. The second enemy gets to execute his decision logic in the next cycle. If he is not in a maze branching position, he cannot change his course.

In order to "match" his position with a multiple of 4 and thus be able to decide which path to take at the fork, we simply start the game with this second enemy placed at a multiple of 4 minus one. Considering coordinates that start at zero, the multiples of 4 are:

First enemy: position 0 or 4 or 8 or 12 or 16 or 20 or 24 or XX (on x or y axis, it doesn't matter)  
 Second enemy: position 1 or 5 or 9 ...

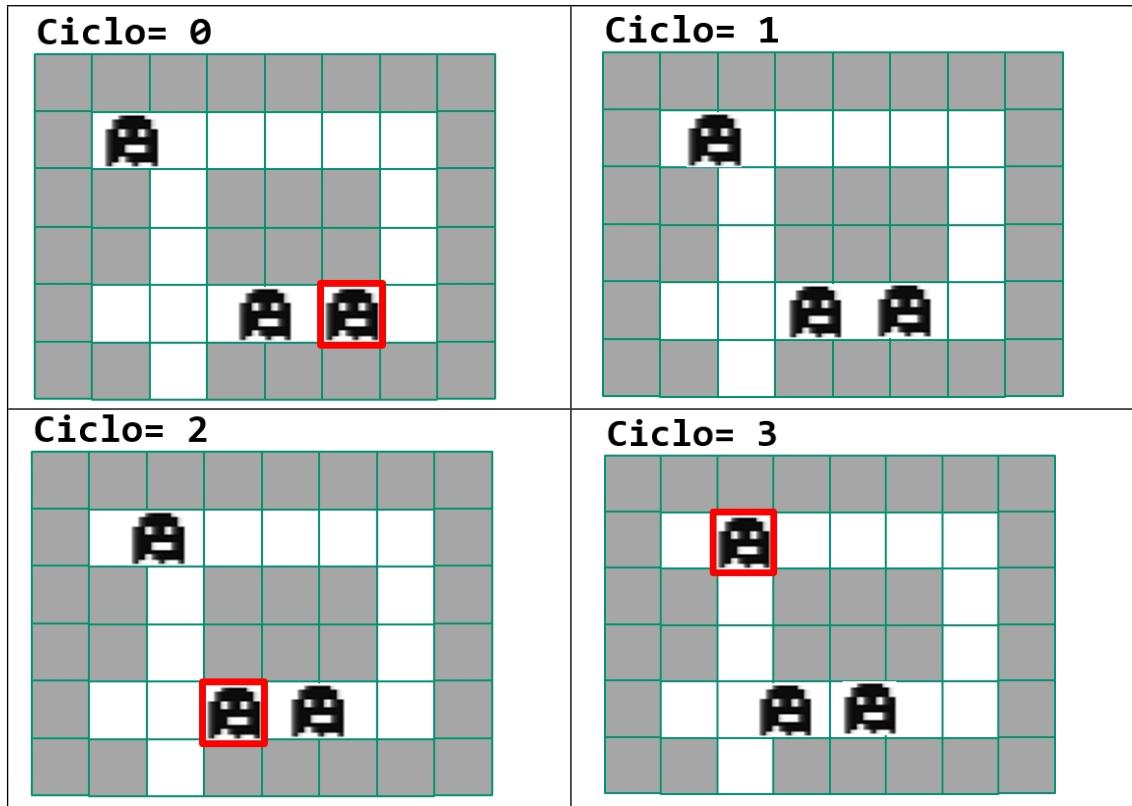
Third enemy: position 2 or 3 or 10 ...

And so on. You place your enemies according to this rule:

**Position = multiple of 4 - n, where n is the sprite number**

And every time it is an enemy's turn to execute his logic, he may find himself at a fork in the road. If a ghost has to make a decision at time "i", then it will make a decision at  $t=i+4$ , and the next decision will be at  $t=i+4+4$ , and so on.

Let's see a graphical example in which I have highlighted with a rectangle that the enemy is going to make a decision because it is in a fork. As you can see, only one fork logic is executed in each game cycle.



*Fig. 66 Massive logics in PACMAN-type games*

When it is your turn to run your logic, you should check that you are not in a position in the middle of a corridor with no branches. You should check this by using the |COLAY command.

The video game "Paco, the man" uses this technique. In fact, it executes the logic of 4 ghosts, each one in a different game cycle, as well as distributing the rest of the tasks among the different cycles. It is a magnificent example of massive logics, so I recommend you to read the "making of" of the videogame, which you will find in the 8BP documentation.

	cycles			
Task	0	1	2	3
Keyboard reading	X			
Detection collision with ghosts y with sprites invisible		X		
Detection collision with layout y relocation if Paco crashes			X	
Detection from points (coconuts)				X
Phantom logic 1	X			
Phantom Logic 2		X		
Phantom Logic 3			X	

The video game "**Paco the man**" distributes the tasks in 4 game cycles, allowing to reach almost 20 FPS in BASIC.

<b>Phantom Logic 4</b>				X	
------------------------	--	--	--	---	--

### 11.3.6 Reducing the number of instructions in the game cycle

Reducing the number of instructions that the game cycle goes through is essential to speed up your program. Sometimes this means that the program will have more lines, even though by end fewer lines are executed in each cycle. Other times you can introduce limitations "undetectable" that speed up your game without the player noticing the difference. Let's take a look at some examples

#### 11.3.6.1 Keyboard management with fewer instructions

Manage the keyboard (and in general this applies to anything you do) by executing the fewest number of instructions. Here is an example (first badly done and then well done), where at most you go through 4 INKEYS operations with their corresponding IFs. Run it mentally and you'll see what I mean. The second one is much faster

Bad example (worst case = 8 "IF INKEY" executions):

```
1000 rem inefficient keyboard routine
1010 IF INKEY(27)=0 and INKEY(67)=0 THEN <instructions>:RETURN
1020 IF INKEY(27)=0 and INKEY(69)=0 THEN <instructions>:RETURN
1030 IF INKEY(34)=0 and INKEY(67)=0 THEN <instructions>:RETURN
1040 IF INKEY(34)=0 and INKEY(69)=0 THEN <instructions>:RETURN
1050 IF INKEY(27)=0 THEN <instructions>:RETURN
1060 IF INKEY(34)=0 THEN <instructions>:RETURN
1070 IF INKEY(67)=0 THEN <instructions>:RETURN
1080 IF INKEY(69)=0 THEN <instructions>:RETURN
1080 IF INKEY(69)=0 THEN <instructions>:RETURN
```

Here is the example of a keystroke reading, but well done (with worst case = 4 "IF INKEY" executions). Also, it has been taken into account that, in an IF, non-zero expressions are TRUE. The instructions to execute in your game may be different but the keyboard reading scheme should be the same in case of handling diagonals (up and right at the same time, for example). It may seem longer, but it is much faster than the previous example.

```
REM efficient keyboard routine 'jump
to 1550 if "P" has not been pressed 1510 if
inkey(27) THEN 1550: 'key P 1520 if
inkey(67) THEN 1530: 'key Q

1525 <instructions in case of having pressed "P" and "Q" at the same time
time>:RETURN
1530 if inkey(69) THEN 1540:'A' key

1535 <instructions in case of having pressed "P" and "A" at the same time
time>:RETURN

1540 <instructions in case you have pressed "P" only>:RETURN 1550 if
inkey(34) THEN 1590:'O' key
```

**1560 if INKEY(67) THEN 1570:'key Q**

**1565 <instructions in case of having pressed "O" and "Q" at the same time time>:RETURN**

**1570 if INKEY(69) THEN 1580: 'key A**

**1575 <instructions if you have pressed "O" and "A" at the same time>:  
RETURN**

**1580 <instructions if you have pressed "O" only>:RETURN**

**1590 IF INKEY(67) THEN 1600:'Q' key**

**1595 <instructions if "Q"> has been pressed: RETURN**

**1600 IF INKEY(69) THEN return:'A' key**

**1610 <instructions if "A" has been pressed only>: RETURN**

Another thing you should do to speed up your game is to use a periodic task to scan "secondary" keys such as keys to activate/deactivate music, keys to switch to a menu or display something special, etc. These are keys that you can scan periodically and not every cycle. However, you have to take into account how much it costs to scan them, which is not much (1 ms).

**10 if inkey(47) then 30: ' this costs 1.0 ms**

**20 <instructions if you press key 47>.**

**30 rem you get here if you have not clicked on it**

Parsing a variable with AND for a task to occur every (for example) 4 cycles costs 1.18 ms, so it is going to cost us

$1.18 \text{ ms} \times 4 \text{ cycles (cycle evaluation)} + 1.0\text{ms (inkey)} = 5.72 \text{ ms}$

if instead of doing that we execute the **inkey** on every cycle, we would have spent only 4ms, therefore, scanning certain keys based on the play cycle only makes sense if we are at least going to avoid scanning in some cycles two keys.

Let's assume the following programme:

**10 if cycle and 3 then 50: ' this costs 1.18 ms**

**20 if inkey(47) ... ' this costs 1 ms**

**30 if inkey(35) ...' this costs 1 ms**

**50 <more instructions>.**

As it is, the program evaluates keys 47 and 35 every 4th cycle. When it evaluates them it spends  $1.18 + 1 + 1 + 1 = 3.8 \text{ ms}$  while when it does not evaluate them it spends 1.18 ms. Therefore, the time spent in 4 cycles is

$\text{Time } 3 * 1.18 + 3.8 = 6.72 \text{ ms}$

Whereas if we had evaluated the keys every cycle we would have spent  $4 * 2 \text{ ms} = 8 \text{ ms}$ . Therefore, there is a saving of  $8 - 6.7 = 1.3 \text{ ms}$  for every 4 cycles, or approximately 0.3 ms per game cycle.

One of the options you have, depending on the type of game, is to scan some keys on even cycles and the others on odd cycles. For example, in a game that uses QAOP keys, you can scan QA on even cycles and OP on odd cycles or vice versa.

That way you can get more speed with a limitation that the player is unlikely to notice. This is the kind of limitation that is sometimes worth introducing, but it depends on the game.

### 11.3.6.2 Avoid going through unnecessary FIs

We will look at two ways of doing this:

```
10 IF A=1 THEN <instructions when A=1> 10 IF A=1 THEN <instructions when  
A=1> 10 IF A=1 THEN <instructions when A=1>  
20 IF A=2 THEN <instructions when A=2> 20 IF A=2 THEN <instructions when  
A=2> 20 IF A=2 THEN <instructions when A=2>  
30 IF A=3 THEN <instructions when A=3>  
40 <more instructions>
```

If you can avoid going through an IF by inserting a GOTO, it is always preferable. The GOTO is a great ally of the massive logic technique.

```
10 IF A=1 THEN <instructions when A=1> : GOTO 40  
20 IF A=2 THEN <instructions when A=2> : GOTO 40  
30 <instructions when A=3> : rem if A is neither 1 nor 2 then it is 3  
40 <more instructions>
```

Another way to do this is by using the ON <variable> GOTO instruction or the ON <variable> GOSUB instruction.

As I have presented in the table of 11.1, you can save more than 1ms by using ON GOTO.

```
10 on A GOTO 30,40,50  
20 goto 60: rem we arrive here if A=4  
30 rem we arrive here if A=1  
35 goto 60  
40 rem we arrive here if A=2  
45 goto 60  
50 rem we arrive here if A=3  
60 rem here the logic continues
```

### 11.3.6.3 Replaces algorithms with pre-calculations

Think of a bouncing ball. Instead of using the equations of accelerated motion, construct a path that moves a sprite downwards with a larger Y-coordinate increment on each step and then upwards with a smaller and smaller increment as it hits the ground. There are no complex equations to run and yet the visual effect is the same. This is how I made the jumps in the game "Fresh fruits & vegetables". This is possible to program this way because **within the game the universe is "deterministic". That is, you can predict at every instant of time what is going to happen**, no matter how complex the equations governing a character's jump or a squad's movement are.

In games where enemy logic requires the use of some function calculation (such as cosine), pre-calculate everything and store it in an array that you use during logic execution. Calculating during game logic is cost prohibitive.

Complex logic is slow logic. If you want to make something complicated, a complex trajectory, an artificial intelligence mechanism... don't do it, try to "simulate" it with a simpler behavioural model but produce the same visual effect. For example, a ghost that is intelligent and chases you, instead of having it make intelligent decisions, have it try to take the same direction as your character, without any logic. If you can't simplify in this way, then think that even artificial intelligence can become "deterministic". If an enemy makes a decision on how to move based on your position and speed, we could store the result of that heavy algorithm in an array and avoid all the calculations.

```
10 Rem Vx, Vy, X, Y are the speed and position of my character.
20 rem let's assume I have pre-calculated decisions for 3 speeds, 10 X-
coordinate slots and 10 Y-slots. This is less than 1KB
30 DIM pursue(3,3,10,100)
40 rem ' load the values into the array after slowly calculating
them
50 newaddress=pursue(Vx,Vy,x,y): rem mechanism in action
```

The universe you are programming is "deterministic". However complex the behaviour of enemies and screen elements may be, if their behaviour does not depend on your interaction, then there is a position for each given thing at each given instant of time. A position that could be pre-calculated to avoid any complex behavioural algorithm and the result would be the same.

#### 11.3.6.4 Do not execute comments

Remove any comments in the game logic and if you leave any that are REM (faster), don't use the quotation mark. If you use the quotation mark it is to save 2 bytes of memory, and is suitable for commenting out the rest of the program (initialisations and such). If you want to comment parts of logic you can do the following:

```
If x>23 gosub 500
...
499 rem is not passed along this line and so I comment on this routine.
500 if x > 50 THEN ...
...
550 RETURN
```

Each comment you execute consumes 0.20 ms and saving its execution is very easy without leaving comments. There are times when you can put comments on lines as long as there are jumps (GOTO and GOSUB/RETURN) without fear of wasting any time, let's see some examples:

```
10 goto 50 : rem this comment is not time-consuming
20 gosub 200: rem this comment is not time-consuming
200 return: rem this comment is not time consuming
```

#### 11.3.6.5 Only one sprite dies in each cycle

The 8BP |COLSPALL command is designed with "massive logic" in mind. This means that it potentially detects the collision of all sprites, but as soon as it detects a collision it returns indicating which sprite is the collider and which is the collided sprite. At that point you can override the colliding sprite (by disabling its ability to be collided in bit 1 of its status byte) and re-run the command, until there are no more collisions (it returns a 32 on the collider and the collided). However,

It is most efficient to not re-trigger the command until the next game cycle. This means that only one enemy can die in each frame, but it is an imperceptible limitation that will greatly speed up your games.

### 11.3.7 Routes that speed up the game by manipulating the state

You can make paths that alternately turn on and off the collider or colliderable flag in your sprites. Depending on the type of enemy, this can give you extra speed in the collision command. Routes are explained in a later chapter, so before reading this, it's a good idea to familiarise yourself with them.

For example, if you have 8 enemies on screen, you can make it so that in the even cycles there are 4 colliderable enemies and in the odd cycles the other 4:

```
ROUTE1;  
-----  
db 1,0,1  
db 255,128+8+2+1,0 ; change of state to colliderable db  
1,0,1  
db 255,128+8+1,0 ; state change to non-collisionable db  
0
```

This path changes the state of your sprite, while moving it to the right. If a sprite has this path assigned to it, it will move to the right and will alternately be collisionable and non-collisionable.

You can assign the path to 8 sprites and then run the following command on 4 of the sprites:

```
|ROUTE8P, <sprite_id>, 1
```

With this, the collision command |COLSPALL will be faster as it will only have to detect collisions with 4 colliding enemies in each frame. This trick speeds up your game a bit and together with other tricks you can finally reach the FPS you need.

The same trick allows you to turn on and off the print flag (bit 0 of the status) and in a path where the enemy spends some frame without moving, you can turn off the print flag through the path and thus speed up the PRINTSPALL command.

### 11.3.8 Routing sprites with "massive logics".

To move sprites through paths there is a command called |ROUTEALL that does this very efficiently, but **as an exercise in understanding the philosophy of bulk logic it is very interesting to study this difficult but emblematic case of bulk logic**. To program the "*Anunnaki*" videogame I used the technique I am going to describe below, since I had not yet programmed the sprite routing capability in 8BP. Basically I used massive logics in the routing of the enemy ships.

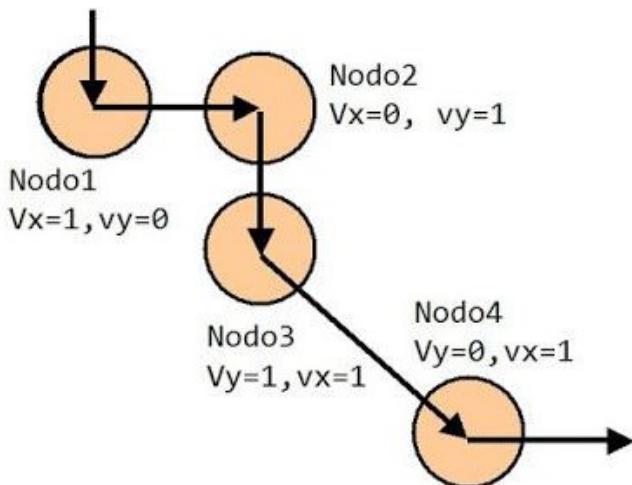
The ships will pass one by one through a series of "control nodes", which are places in space where they must change their direction, defined by their velocities in X, and Y, i.e. (Vx,Vy).

One way to control that the 8 ships change direction at these locations would be to compare their X,Y coordinates with that of each of the control nodes and if they coincide with any of them, then we apply the new velocities associated with the

change in that node. Since we are talking about 2 coordinates, 8 ships and 4 nodes, we are looking at:

$$2 \times 8 \times 4 = 64 \text{ checks on each frame}$$

This is not feasible if we want speed from BASIC, as it is not a computationally efficient strategy. Since we are dealing with a "**deterministic**" scenario, we can be sure at each instant of time where each of the ships are going to be and therefore instead of checking in space, we can only **focus on the time coordinate** (which is the game frame number or the so-called "game cycle" number). Do not consider time in seconds, but in frames.



*Fig. 67 Defined trajectory with "control nodes".*

Since we know the speed at which the ships are moving, we can know when the first ship will pass the first node. We will call that instant  $t(1)$ . We will also assume that because of the separation between the ships, the second of the ships will pass the node at the instant  $t(1)+10$ . The third at  $t(1)+20$  and the eighth at  $t(1)+70$ . Instead of using frames as time units, let's use tens of frames: in that case the instants will be  $t(1)$ ,  $(1)+1$ ,  $t(1)+2$ , etc.

Knowing this we can control the time with two variables: one will count the tens (i) and the other the units (j). To control the change of the 8 ships in the first node we can write:

```
j=j+1: IF j=10 THEN j=0: i=i+1: IF i>=t(1) AND i<=t(1)+8 THEN
[updates ship's speed]           i-t(1) with the velocity values of node
1.]
```

As we can see, with a single line we can change the velocities of each ship as they pass through node 1. Each time "j" becomes zero, we increase the variable "i" and update one of the ships. During the first 80 instants of time (8 in tens of frames) each of the 8 ships are updated, just as they pass through the control node, i.e. at instant  $t(1)$  Sprite 0 is updated, at  $(t1)+1$  Sprite 1 is updated, at  $t(1)+2$  Sprite 2 is updated, and so on.

The Sprite number that appears on the line is  $i-t(1)$ , so if  $t(1)=1$  we want to be the frame 40, ( $t=4$ ) then when "i" is 4 will start to update Sprite 0, and when "i" is 11 will update Sprite 7 (8 ships in total).

Now let's apply the same to all 4 nodes. We could run 4 checks instead of one, but it would be inefficient. Also, if we had a lot of nodes this would mean a lot of checks. We can do it with just one, taking into account that the first ship passes through a node at an instant  $t(n)$  and the eighth ship passes through that node at  $t(n)+7$ .

When the first ship passes through the first node, it makes sense to think about starting to check node 2, but not node 3 or node 4.

As for the smallest node, we can assume that, even if we have 20 nodes, they are far enough apart so that there are no ships traversing more than 3 nodes at a time (we will assume that and use that "3" as a parameter). Therefore, the smallest node to check is the largest - 3. We will call the smallest node "nmin" and the largest "nmax" ( $nmin = nmax-3$ ). In case we want to have full freedom to define any trajectory, nmin must be nmax minus the number of ships in the row.

```
10 j=j+1: IF j=10 THEN j=0: i=i+1: n=nmax  
20 IF n<nmin THEN 50: ' no more ships to be updated  
30 IF i>=t(n) AND i<=t(n)+8 THEN [update i-t(n) ship with node speeds  
n]:IF i-t(n)=0 THEN nmax=nmax+1: nmin=nmax-3  
40 n=n-1
```

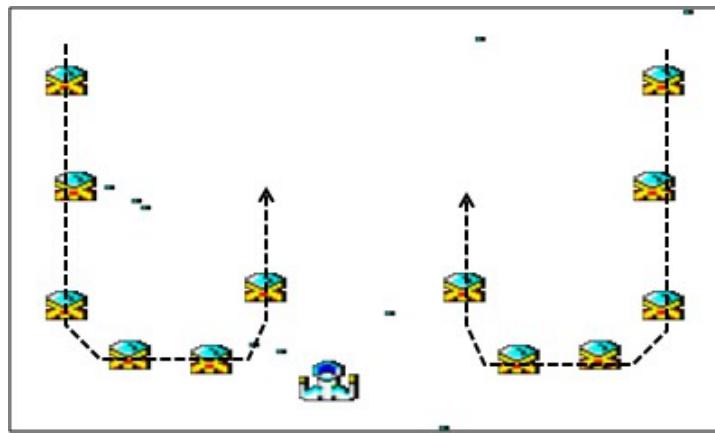
#### 50 ' plus game instructions

As you can see, when the time decade (variable "i") is incremented by 1, it starts checking if there is any ship in one of the nodes from "nmax" to "nmin", updating only one ship in each frame. If the ship being updated is zero, then the maximum node is incremented, as that ship is on its way to the next node.

For the next frame, the number of nodes is decreased (instruction  $n = n-1$ ) so that we will check if there is a ship in the previous node, and so on until nmin. Always, however, checking only one ship in each frame.

In short, we have transformed 64 checks into just 1, using "Mass Logic". And if the path had 40 instead of 4 nodes, we would have transformed 640 operations into one!

The video game "**Annunaki**" uses this technique to manage the trajectories of two symmetrical rows of 6 ships each. It's complicated, but as you can see, from BASIC you can take control of 12 ships and make them move along whimsical trajectories, using more intelligence than power, thanks to the massive logic technique.



*Fig. 68 Two rows with massive logics*

## 12 Complex trajectories: ROUTEALL command

This is an "advanced" command available since version V25 of the 8BP library. It simplifies programming a lot because you can define a path and make a sprite go through it step by step using the ROUTEALL command.

First, you need to create a route. To do this you need to edit it in the routes\_yourgame.asm file.

Each route has an indeterminate number of segments (although the maximum length of a route is 255 bytes) and each segment has three parameters:

- How many steps we are going to take in that segment (between 1 and 250)
- What velocity Vy is to be maintained during the segment ( $-127 \leq Vy \leq 127$ )
- What speed Vx is to be maintained during the segment ( $-127 \leq Vx \leq 127$ )

As a route can be at most 255 bytes long and a segment occupies 3 bytes, a route can have at most 84 segments.

At the end of the segment specification we must put a zero to indicate that the path has been terminated and that the sprite should start traversing the path from the beginning. Let's look at an example:

```
; LIST OF ROUTES
;=====
; Put here the names of all the routes you make
ROUTE_LIST
    dw ROUTE0
    dw ROUTE1
    dw ROUTE2
    dw ROUTE3
    dw ROUTE4
    dw ROUTE4

; DEFINITION OF EACH ROUTE
;=====
ROUTE0; a circle
;-
    db 5,2,0; five steps with Vy=2
    db 5,2,-1; five steps with 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; left-right
;-
    db 10,0,-1
    db 10,0,1
    db 0

ROUTE2; up-down
;-
    db 10,-2,0
    db 10,2,0
```

```

db 0

ROUTE3; one
eight
;-----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; a loop and goes to the left**

```

;-----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

```

Now to use the routes from BASIC, we simply assign the route to a sprite with the SETUPSP command, indicating that we want to modify parameter 15, which is the one that indicates the route. In addition, we have to activate the route flag (bit 7) in the status byte of the sprite and we will set it with the automatic movement flag and the animation and print flags.

```

10 MEMORY 24999
11 ON BREAK GOSUB 280
20 MODE 0:INK 0,0
21 LOCATE 1,20:PRINT "command |ROUTEALL and animation
macrosequences".
30 CALL &6B78:DEFINT a-z
31 |SETLIMITS,0,80,0,200
40 FOR i=0 TO 31:|SETUPSP,i,0,0,0: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 with this activate the route flag
70 |SETUPSP,i,7,2:|SETUPSP,i,7,33: rem macro animation sequence
71 |SETUPSP,i,15,3: rem assigned route number 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 here AUTOALL already invokes ROUTEALL
120 GOTO 100

```

## 280 |MUSIC:MODE 1: INK 0,0:OPEN 1

We've got it all. This advanced technique will simplify your programming a lot with spectacular results.



Fig. 69 an 8-shaped route

As you have seen, the command does not modify the coordinates of the sprites, so they must be moved with **|AUTOALL** and printed (and animated) with **|PRINTSPALL**. That's why you have an optional parameter in **|AUTOALL**, so that **|AUTOALL,1** internally invokes **|ROUTEALL** before moving the sprite, saving you a BASIC invocation that will always take a precious millisecond.

### 12.1 Places a Sprite in the middle of a route : ROUTESP

If you assign several sprites to the same route and you want them all to go in single file as in the example above, then you need to make sure that each sprite is at a different point on the route. There are two ways to do this

The first one consists of gradually assigning the route to the sprites. That way, the sprite in the lead is the first to be routed and after a few game cycles you route the next one and then the next one, and so on. In each cycle you execute **|AUTOALL,1** and that routes the sprites that already have a route assigned to them, which will be ahead in the number of steps with respect to the sprites that don't have it assigned yet.

The second option is to use the **|ROUTESP** command.

#### |ROUTESP, <spriteid>, <steps>.

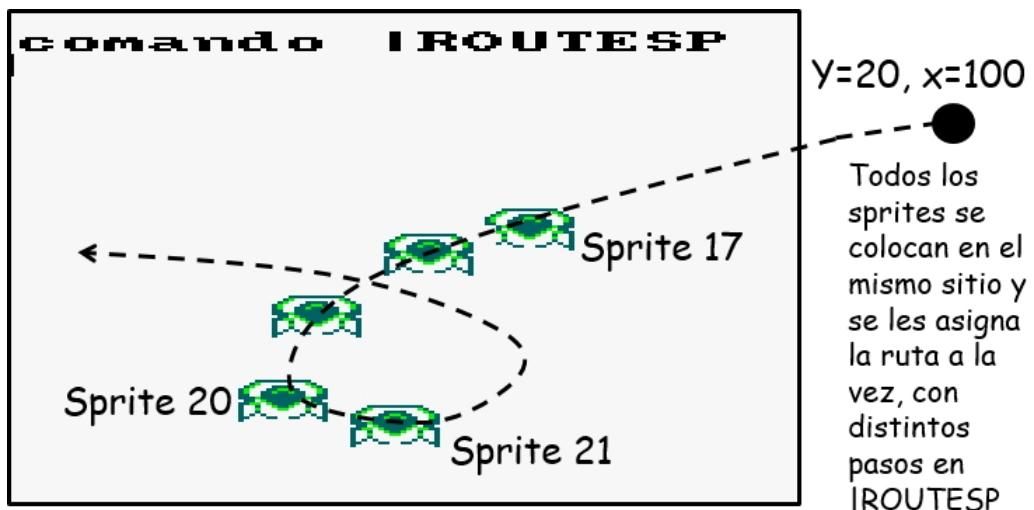
This command moves a sprite the desired number of steps (**up to 255 steps**) along its assigned path. In the example I have highlighted in red the assignment of path 8 and the **|ROUTESP** commands that position each of the sprites along the path.

```
10 memory 24999
10 ON BREAK GOSUB 12
11 GOTO 20
12 |MUSIC:CALL &BC02:PAPER 0:OPEN 1:MODE 1:END
20 CALL &BC02:DEFINT A-Z:MODE 0
50 CALL &6B78
70 ' all vessels placed at the same initial co-ordinate
75 ' and with the same route but with a different initial number of steps.
```

```

76 locate 2,3: Print "command |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:'ship head
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,0,0,0,0
140 --- game cycle ---
150 |AUTOALL,1: |PRINTSPALL
160 goto 150

```



*Fig. 70 Row houses through the use of ROUTESP*

Route 8 would be defined in the routes\_mygame.asm file as follows:

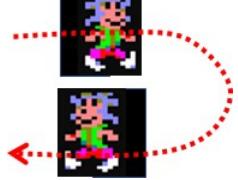
```

ROUTE8;
    db 255, 128+64+32+32+8+1,0; state change
    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
;
repositioning
    db 1,0,115
    db 1,-30,0
    db 120,0,0
    db 120,0,0
    db 0

```

## 12.2 Creating Advanced Routes

There are 4 functionalities that you can use in the middle of any route (**note, in the middle, not at the end**), using an escape code as the value of the number of steps of a segment:

Escape code	Description	Example
<b>255</b>	Change of sprite state.	DB 255, 3, 0 State goes to value 3. The zero at the end is a filler.
<b>254</b>	Change of sprite animation sequence  After changing the sequence, if you want the image to change as well, you must use the code 251	DB 254, 10, 0 The sequence 10 is associated. The zero is a filler If the assigned sequence is the one the sprite already has, then it is harmless (no reset of the frame id). In case you want to reset the frame id, the third parameter must be a 1 , for example: DB 254, 10, 1
<b>253</b>	Change of image 	DB 253 DW new_img The image "new_img" is associated, which must be a memory address.
<b>252</b>	Change of route	DB 252,2,0 Route 2 is associated
<b>251</b>	Go to next frame from the animation. 	DB 251,0,0 The Sprite is animated. The two zeros are fillers

**IMPORTANT:** be very careful to write DB and DW where they should be used, i.e., for example, if you change images you should precede the image with DW and not with DB. If you make such a mistake, your route will not work.

**IMPORTANT:** escape codes can be used in the middle of a route, but the last segment cannot be an escape code, it must be a movement, even if it is standing still, something like "DB 1,0,0".

### 12.2.1 Forced state changes from routes

This ability is very interesting to speed up your games and is available since V27. It means that we can force a change of state in the middle of a route. To do this, we indicate that we want a change of state by indicating the number of steps in the segment as a value = 255.

The status change is one more segment and it is important that you keep the same number of parameters per segment, i.e. 3 bytes. A status change to status=13 could be written as:

**255,13,0**

The third parameter (the zero) does not mean anything, it is just a "filler" to make the segment measure 3 bytes, but it is essential.

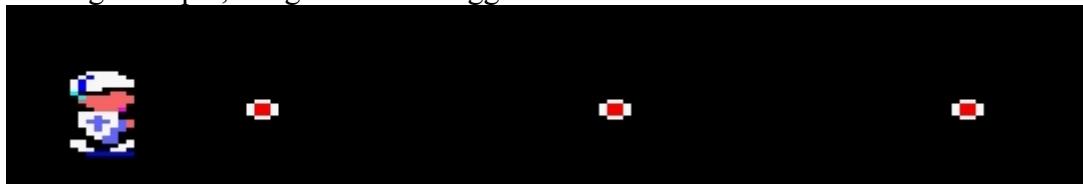
The value 255 will tell the **|ROUTEALL** command that what it should do this time is to change the state of the sprite, assigning it the state indicated below. The state change is executed without consuming a step, so the next step after the state change will always be executed. If we don't want the sprite to move anymore, we simply define a one-step segment with no movement in X or Y right after the state change. Let's see an example:

```
ROUTE3; fire_dere
;-----
db 40,0,2; forty steps to the right with Vx=2 db
255,0,0; change of state to zero
db 1,0,0; still Vy=0, Vx=0 db 0

ROUTE4; trigger_izq
;-----
db 40,0,-2; forty steps to the left with Vx=-2 db
255,0,0; change of state to zero
db 1,0,0; still Vy=0, Vx=0 db 0
```

We are going to use these two routes to shoot with our character. The first one, after going through 40 steps in which it advances 2 bytes in X, undergoes a state change and the sprite goes to state 0, i.e. deactivated. The next segment has only one step and no movement (vy=0, vx=0).

With this mechanism we can trigger and have the triggers deactivate themselves when they leave the screen. This saves BASIC logic and speeds up our games. In the following example, image 26 is the trigger.



*Fig. 71 Triggers with en-route status change*

```
10 MEMORY 24999
20 MODE 0: DEFINT A-Z: CALL &6B78: 'install RSX
25 ON BREAK GOSUB 280
30 CALL &BC02: 'restore default palette just in case'.
40 INK 0,0: 'black background'
50 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT: 'reset sprites'
80 |SETLIMITS,12,80,0,186: ' set the limits of the game screen
90 x=40:y=100: ' character coordinates'
100 |SETUPSP,0,0,1: ' character status'
110 |SETUPSP,0,7,1:dir=1: 'animation sequence assigned on start'
120 |LOCATESP,0,y,x: 'place the sprite (without printing it yet)
```

```

125 |MUSIC,0,0,6
126 for i=1 to 4:|SETUPSP,10+i,9,26:next:'shots
130 cycle of play---', 'cycle of play---
150 |AUTOALL,1:|PRINTSPALL,0,0
170 ' character movement routine -----
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 wait<cycle-10 then if INKEY(47)=0 THEN wait=cycle: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 cycle=cycle+1
210 goto 150
280 |MUSIC:MODE 1: INK 0,0:PEN 1

```

State changes can be forced at any segment of the route, not necessarily at the end, although in the case of a trigger it is very logical to do so at the end of the route.

### 12.2.2 Forced sequence changes from routes

We can change the animation sequence of a sprite using a special segment. When we put 254 in the value of the number of steps, the ROUTEALL command will interpret that an animation sequence change should be performed on the sprite. Example:

**254,10,0**

This segment changes the animation sequence of the sprite, setting the sequence number 10. The third parameter (the zero) means that the sequence will not be restarted, so if the sprite is in frame 5 of another sequence, the new sequence will be assigned and disco frame will be maintained, although it will now correspond to another image. This is very useful because we can assign a sequence to a sprite within a path and if it already had it, it is harmless. In order to reset the assigned sequence (to frame 0) you have to put a 1 in the third parameter:

**254,10,1**

In case you assign a sequence and want it to restart, it is advisable to use the animation code, the 251 that we will see later. That way, the image associated to the sprite will change in the sprite table and not only the **frame\_id**.

As with the state change, the sequence change is executed without consuming a step, so the next step to the sequence change will always be executed.

### 12.2.3 Forced image changes from routes

We have seen how to route sprites with ROUTEALL or even better, with AUTOALL,1

Often we don't want to route a sprite through a trajectory but something more everyday: jumping with a character. In the example in chapter 9 we saw how to do this with a BASIC array containing the relative movements of the Y-coordinate. In this case we are going to do the same with a path, achieving a faster movement.



*Fig. 72 Skip using a route*

To avoid having to check if the character has reached the zenith point of the jump, we can use a special segment that indicates image change. Like any other segment, it consumes 3 bytes, but in this case the first one is the image change indicator (a 253 value) and the next two correspond to the memory address of the image. **BE CAREFUL**, you must use "**dw**" before the name of the image you want to allocate, so you will have to write this image change segment in two lines. A "**db**" for the 253 and a "**dw**" for the image memory address.

```
db 253
dw SOLDIER_R1_UP
```

In the following example we have a jumping dummy. On the way up the dummy erases itself at the bottom, while on the way down it erases itself at the top. So that there is no discontinuity in the movement, just when changing one image for another it is necessary to vertically realign the soldier, raising the down image exactly 5 lines, to make it coincide with the soldier that was going up.

This would be the file sequences\_mygame.asm

```
;=====
; up to 31 animation sequences
;=====
must be a fixed table and not a variable table
;b each sequence contains the addresses of cyclic animation frames
Each sequence is 8 image memory addresses.
even number because the animations are usually an even number
a zero means end of sequence, although 8 words are always spent.
When a zero is found, a new start is made.
if there is no zero, after frame 8 it starts again.

The zero sequence is that there is no sequence.
We start from sequence 1
```

```
;-----animation sequences -----_
SEQUENCES_LIST
dw SOLDIER_R0,SOLDIER_R2,SOLDIER_R1,SOLDIER_R2,0,0,0,0,0 ; 1
dw SOLD_L0,SOLD_L2,SOLD_L1,SOLD_L2,0,0,0,0,0 ; 2 dw
SOLD_R1_UP,0,0,0,0,0,0,0;3
dw
SOLDIER_R1_DOWN,0,0,0,0,0,0,0,0;4
dw SOLDIER_L1_UP,0,0,0,0,0,0,0,0;5
dw SOLDIER_L1_DOWN,0,0,0,0,0,0,0,0;6
```

**\_MACRO\_SEQUENCES**  
**;-----MACRO SEQUENCES -----**  
**are groups of sequences, one for each direction.**  
**; the meaning is:**  
**; still, left, right, up, up-left, up-right, down, down-left, down-right**  
**The numbers are numbered from 32 onwards**  
**db 0,2,1,3,5,3,3,4,6,4; 32 --> soldier sequences , id=32. next would be**  
**33**

We will use two routes, one to jump right and one to jump left. This would be the file routes\_mygame.asm

```
; LIST OF ROUTES  

;=====  

; put here the names of all the routes you make  

ROUTE_LIST  

dw ROUTE0  

dw ROUTE1  

dw ROUTE2  

dw ROUTE3  

dw ROUTE4  

dw ROUTE4

DEFINITION OF EACH ROUTE  

;=====  

ROUTE0; jump_right  

db 253  

dw SOLDIER_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 SOLDIER_R1_DOWN  

db 1,-5,1; up so that UP and down match db  

2,1,1  

db 2,2,1  

db 2,3,1  

db 2,4,1  

db 1,5,1  

db 253  

dw SOLDIER_R1  

db 1,5,1; go down one more, as R1 has no blacks on top  

db 255,13,0; new state, without path flag and with animation flag db  

254,32,0; macro sequence 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 SOLDIER_L1_DOWN  

db 1,-5,-1; raise to fit UP and down db 2,1,-1
```

```

db 2,2,-1
db 2,3,-1
db 2,4,-1
db 1,5,-1
db 253
dw SOLDIER_L1
db 1,5,-1; down one more
db 255,13,0; new state, without path flag and with animation flag db
254,32,0; macro sequence 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; fire_dere
;-----
db 40,0,2
db 255.0
db 1,0,0
db 0

ROUTE4; trigger_izq
;-----
db 40,0,-2
db 255.0
db 1,0,0
db 0

```

And this would be the example programme. Compare its execution with the one in chapter 7 to see the difference in speed. You will notice a big difference

```

10 MEMORY 24999
20 MODE O: DEFINT A-Z: CALL &6B78:' install RSX
25 ON BREAK GOSUB 2800
30 CALL &BC02:ink 0,0:'restores default pallet
50 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:'reset sprites
80 |SETLIMITS,12,80,0,186: ' set the limits of the game screen
90 x=40:y=100:jump=0:cycle=40:' coordinates of the character
100 |SETUPSP,0,0,13:|SETUPSP,0,5,0,0:' character status
110 |SETUPSP,0,7,1:|SETUPSP,0,7,32:'animation sequence assigned on start

120 |LOCATESP,0,y,x:'we place the sprite      print it yet)
    (without
123 locate 1,1: print "press Q" : print "stop" skip": print "example
    : print "for
with route"
124 print "press SPACE to fire" print "press
    SPACE to fire" print "press SPACE to fire"
    print "press SPACE to fire" print
125 PLOT 1,150:DRAW 640,150: PLOT 92,150:DRAW   92,400: 'floor and wall
126 for i=1 to 4:|SETUPSP,10+i,9,26:next:'shots
127 |MUSIC,0,0,5: 'music starts playing
130 ----- 'cycle of play'.

```

150 |AUTOALL,1:|PRINTSPALL,0,1,0  
170 ' character movement routine -----

```

172 IF INKEY(47)=0 THEN if wait<cycle-10 then wait=cycle: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: ' if state is
>128 is that I'm jumping (has route)
174 IF INKEY(67)=0 THEN |SETUPSP,0,0,137:|SETUPSP,0,15,dir:'skip
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 left
193 cycle=cycle+1
310 goto 150
2800 |MUSIC:MODE 1: INK 0,0:PEN 1

```

To check that the dummy is jumping I simply query the status by peek (27000). This way, we will know if it has the routing flag active and if so, we will not go through the lines that move it left and right.

#### **12.2.4      Forced rerouting from routes**

We can change the path of a sprite using a special segment. When we put 252 in the value of the number of steps, the ROUTEALL command will interpret that a path change should be made to the sprite. Example:

**252,2,0**

This segment changes the path of the sprite, setting the path number 2. The third parameter (the zero) means nothing, it is just a "filler" to make the segment measure 3 bytes, but it is essential.

As with the state change, the route change is executed without consuming a step, so the next step after the route change will always be executed, which will be the first step of the first segment of the new route.

Since a route can be at most 255 bytes long and a segment is 3 bytes long, a route can be at most 84 segments long. You may need to build an even longer route, and in that case you can do so by concatenating the end of a route with a change of route to another route, and you can concatenate as many routes as you wish.

#### **12.2.5      Forced path changes from BASIC**

Suppose you want your character to jump to the right and in the middle of the jump you want the character to be able to change to the left, continuing the jump. What we are proposing can be represented with this drawing:

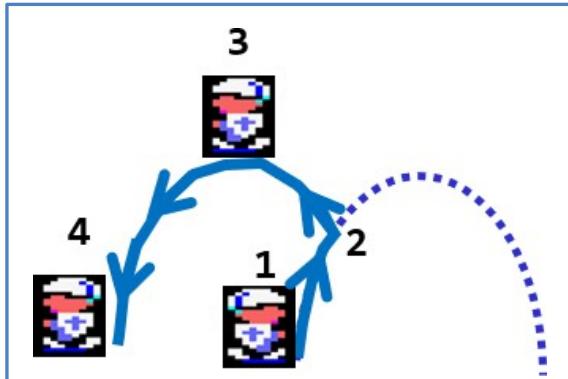


Fig. 73 Change of route in the middle of a jump

In this example our character jumps and in the middle of the jump path to the right, he changes direction at point 2, continuing the jump path to the left, but without starting a new jump, simply continuing the jump and reaching the same height (point 3) that he would reach with the jump to the right to finally finish the jump to the left at point 4.

In order to do this we need to change the path to our character from BASIC at point 2, but we cannot use the **|SETUPSP** command because that would initialise the path, resulting in a much larger jump, starting at point 2. What we can do in that case is simply **POKE** to the memory address that stores the path of the Sprite, which is the address of its +15 state, as shown in the sprite attribute table. This **POKE** changes the path, but keeps the position intact (segment number and position where the character is located). WARNING: the two paths must have the same segments and length, because if you jump to a shorter path and find yourself on a segment that does not exist on that path, an unpredictable effect may occur.

Assuming we have two jump paths (path 0 jump right and path 1 jump left), the following example illustrates the concept:

```

130 ----- 'cycle of play'.
150 cycle=cycle+1:AUTOALL,1:|PRINTSPALL,0,1,0
170 'character movement routine -----
173 IF PEEK(27000)<128 THEN 178
174 we are in the middle of a leap
175 IF INKEY(27)=0 THEN POKE 27015,0:GOTO 180:'change path to the right
176 IF INKEY(34)=0 THEN POKE 27015,1:GOTO 180:'change path to left
177 GOTO 150
178 IF INKEY(67)=0 THEN |SETUPSP,0,0,137:|SETUPSP,0,15,dir:'skip
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 left
310 GOTO 150

```

The only limitation of using **POKE** for this purpose is that the character's picture does not change until it encounters a picture change code in the middle of the path. Changing the image using **|SETUPSP** is possible but dangerous, as you don't know whether the character is going up (erased with lower lines) or down (erased with upper lines). So it is better to simply assign the path and have the path itself change the image as soon as possible. You can even put in the middle of the route image changes, even if they are not necessary in case this circumstance occurs. What you do have to make sure is that the jump image has deletion bytes on both sides, because if it doesn't and changes direction, it will leave a trace.

### 12.2.6      Forced animation from routes

We can leave the animation flag of a Sprite inactive and animate it only at certain instants of the path using code 251:

**251,0,0**

This can be very useful for giving a sense of an approaching sprite in games that use pseudo-3D techniques. For example, for an approaching meteorite that we want to change frames to appear larger. The meteorite will move a few times before moving to the next size. This mechanism is very similar to the image change mechanism, except that it allows you to define the image change without explicitly specifying the image, but simply indicating a frame change in the animation sequence assigned to the sprite.



*Fig. 74 Forced animation from route*

With this flag you can use the same route for an approaching space bird or a meteorite. By not indicating the image, in each case the image that corresponds to the sequence that each sprite has will be applied.

### 12.2.7      How to build "dynamic" (not predefined) routes

A dynamic route is a route whose path is decided at runtime in your BASIC program. It is useful when your program dynamically generates mazes or race tracks that an enemy must run through and which are not known a priori and therefore cannot be defined in the "routes\_mygame.asm" file.

In order to make such a route and assign it to a Sprite, what we have to do is to create an "empty" route in the file "routes\_mygame.asm", in this case route 2.

```
; LIST OF ROUTES
;=====
Put here the names of all the routes you make
ROUTE_LIST
    dw ROUTE0
    dw ROUTE1
    dw ROUTE2
    dw ROUTE3
    dw ROUTE4
    dw ROUTE4
```

#### DEFINITION OF EACH ROUTE

```
;=====
```

```

ROUTE0; right left db 10,0,1
  db 10,0,-1
  db 0

ROUTE1; up down db
  10,-1,0
  db 10,1,0
  db 0

ROUTE2; dynamic routing
  ds 100

```

We have created path 2 with 100 bytes free to fill from BASIC. It may be that the path we build will take up less and in that case it will be enough to reserve fewer bytes. Once the library and the graphics have been assembled, we must look for the memory address of the "ROUTE2" label in the winape symbol window. When we have it, from BASIC we will program the route using POKE from that memory address and the following ones

POKE puts a byte into a memory address. Our numbers must belong to the range -127..128 and POKE does not allow to put negative numbers. To do this you must use the positive value with which the Amstrad internally represents negatives (i.e. the two's complement). To do this you just need an AND 255 operation

<b>10 A=-10</b>
<b>20 PRINT A: REM this prints a 10</b>
<b>30 PRINT A AND 255: REM this prints a 246 which is the same -10</b>

In short, if you want to insert a -10 you have to insert a 246 and use the same strategy for any negative number. Don't forget that the last POKE of the route must be the insertion of a zero, which means end of route.

### 12.2.8 Programming of routes including patterns

Let's assume you want to make a path for an enemy to traverse the screen from right to left over and over again, assuming we start from an initial position where the sprite is on the far right of the screen.

<b>ROUTE0; single route</b>
<b>DB 80,0,-1 ; 80 steps. In each step 1 byte is moved</b>
<b>DB 1,0,80 ; reposition the sprite to its original position DB</b>
<b>0</b>

This path moves a sprite with a 1-byte step every frame. If we wanted to slow it down, moving 1 byte every other frame, we could do the following:

<b>ROUTE0; not so simple route DB</b>
<b>1,0,-1</b>
<b>DB 1,0,0</b>
<b>DB 0</b>

Now this route allows us to move a sprite more slowly, but we cannot reposition the sprite to its original position. This is because as the screen is 80 bytes wide, we need 160 segments, as the sprite advances 1 byte every two segments. The resulting path would be very long and in fact we would have to concatenate 2 paths because a path can only measure 255 bytes (84 segments).

The optimal solution is to define a short path without repositioning the sprite and reposition it from BASIC, using something like this:

```
80 |SETUPSP,31, 0, 128+16+1: REM routable, automatic mov, printable
85 |LOCATESP,31,100,80:' placed on the right side of the screen.
90 rem game cycle
100 cycle=cycle +1
110 |AUTOALL,1: |PRINTSPALL
120 if MOD cycle 160=0 THEN |LOCATESP,31,100,80: ' repositioning
130 goto 100
```

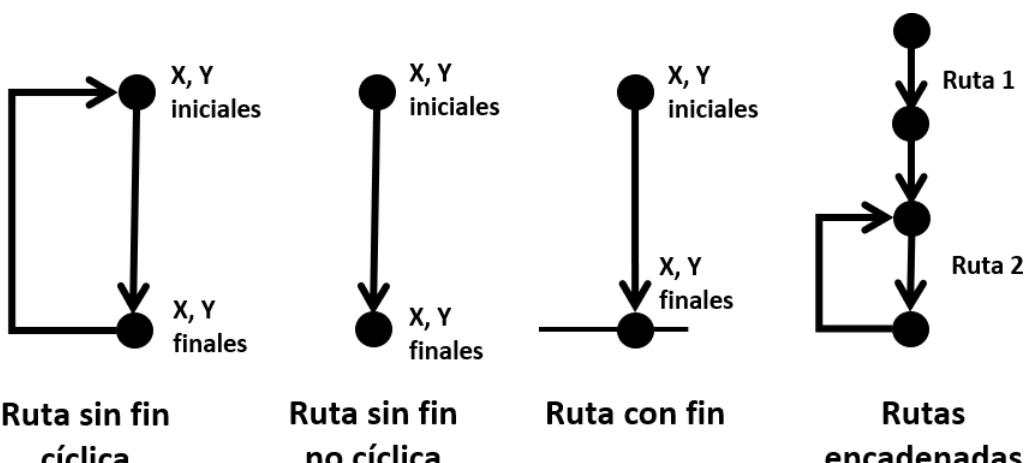
This kind of strategies are useful whenever we don't want to repeat the same movement in every frame, but we want to define a repeating pattern in which in certain frames the sprite moves in one direction and in others it moves in another direction or even not at all. Let's look at another example, an inclined path in which for every 3 vertical movements we move one horizontal movement.

```
ROUTE0; inclined route
DB 2,1,0
DB 1,1,1,1
DB 0
```

### 12.2.9 Typology of routes

With all that we have seen we can classify the routes into the following types:

- **Cyclic endless paths:** they reposition the sprite or simply end up at the same coordinates where they started.
- **Non-cyclic endless paths:** they advance indefinitely and do not reposition the sprite so they can move infinitely away from the play area, unless we reposition the sprite from BASIC.
- **Routes with end:** in the last step change the state of the sprite by disabling the routing flag.
- **Chained routes:** from one route you can jump to another route and this second route can be cyclic or non-cyclic or have an end, or even end up jumping to a third route.

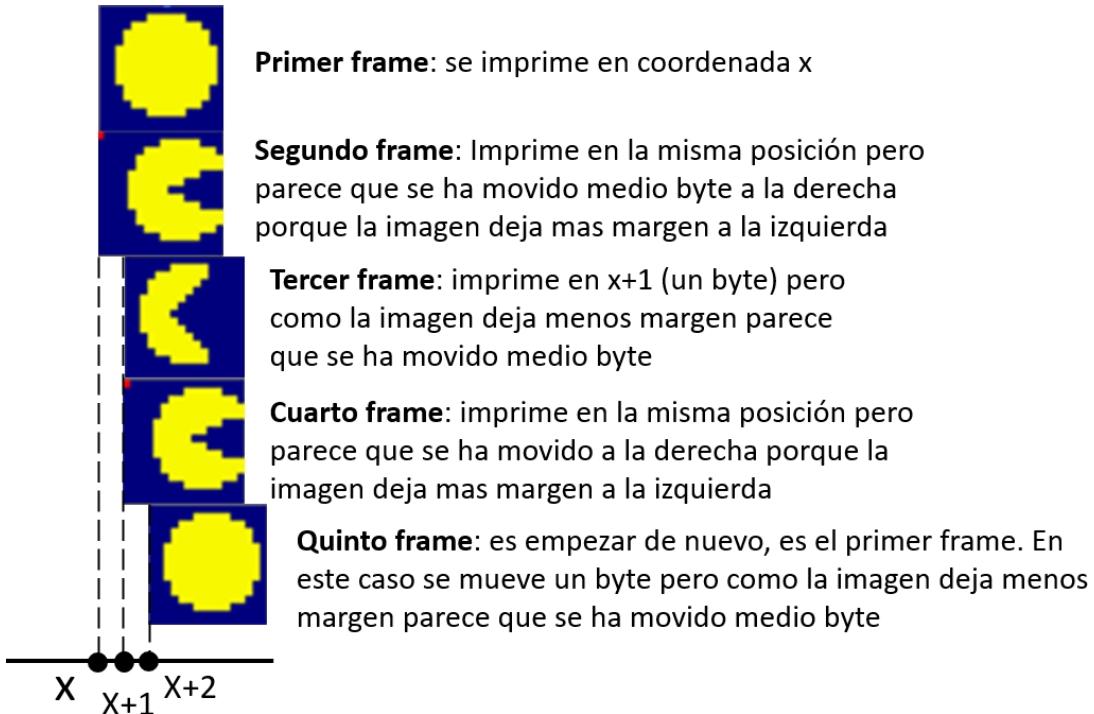


*Fig. 75 Types of routes*

## 13 Half-byte smooth motion

8BP moves the sprites byte by byte with commands like MOVE, and its coordinate system is bytes, not pixels. Therefore, we are talking about 80 positions on the horizontal axis and 200 on the vertical axis.

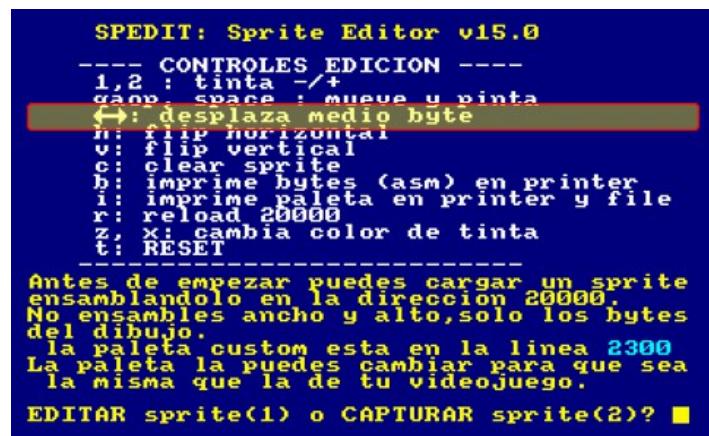
A byte contains 2 pixels in mode 0, or 4 pixels in mode 1. We may want a smoother movement (pixel by pixel in mode zero or two pixels in mode 1). There is a simple trick we can use for this. It is to have an image of the character shifted by half a byte and simply assign it to the Sprite. Even if it is printed at the same coordinate, it will look like it has moved.



In this animation sequence, there are times when the Pac-Man does not move, but as we change the image, it is as if it moved 2 pixels (half a byte). At the times when it moves 1 byte, the image appears shifted to the left half a byte, so the "net" result is also as if it moved 2 pixels (half a byte). To define the movement of this Pac-Man we can use the 8BP path mechanism. This would be the example of the path of movement to the right:

```
ROUTE0; right 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
```

Drawing displaced images can be a bit tedious. That's why since **SPEDIT** version **V15** you have a mechanism to shift an image half a byte (2 pixels in mode 1 or one pixel in mode 0) to the right or to the left As you can see in the SPEDIT menu a new option appears: with the arrow keys you can shift the image you have drawn



```
SPEDIT: Sprite Editor v15.0
---- CONTROLES EDICION ----
1,2 : tinta -/+ 
space : mueve y 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)? █
```

This allows us to easily move an image to apply the described technique of smooth movement.

## 14 Scrolling games

The 8BP library allows you to scroll in different ways that can be combined simultaneously, although the most important method is based on the **|MAP2SP** command. The available techniques are summarised below:

- **By means of sprite block movement commands:** A simple way to scroll with 8BP is to simply create decorative sprites that we set their state to be moved by the **|MOVERALL** and/or **|MOVEALL** commands.  
**|AUTOALL.**
- **Using |MAP2SP:** The idea behind the multidirectional scrolling provided by **|MAP2SP** in 8BP is simple: all the elements that are represented on screen are sprites, so the elements of the world that we are going to print and move around the screen are sprites whose associated images will be mountains, houses, trees or whatever you need to build your "world". To select a portion of the world and transform it into a list of sprites, the **|MAP2SP** function is used. The **|UMAP** function allows you to update the world with a portion of a larger world.
- **Using the |STARS command, which will** allow you to make multidirectional scrolling of a bank of 40 pixels that you can place wherever you want and that you can move in different planes and at different speeds.
- **Using the |RINK command,** which will allow you to rotate an ink pattern, giving a sense of forward motion that you can use in certain types of scroll, such as movement of a brick floor, water, etc.

### 14.1 STARS: Scroll of stars or mottled earth

In the 8BP library you have a very easy to use function to create a background effect of moving stars, giving the sensation of scrolling. This is the function **|STARS**. This function is able to move up to 40 stars simultaneously without altering your sprites, so it is as if they pass "underneath".

**|STARS,<initial star>,<num stars>,<colour>,<dy>,<dx>,<dx>.**

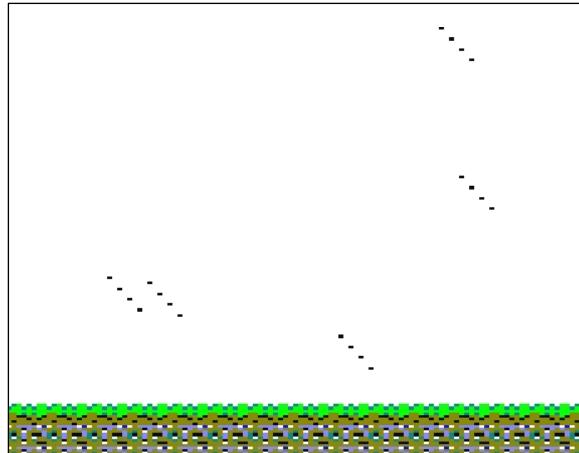
You have a bank of stars and can combine several STARS commands to work with groups of stars at different speeds, giving a sense of planes with different depths.

The star bank consists of 40 pairs of bytes representing (y,x) coordinates. Occupying from address 42540 to 42619 (80 bytes in total). One way to generate 40 random stars would be (note that if we have already executed DEFINT A-Z the number 42540 must be put in hexadecimal because it is greater than 32768).

```
FOR dir=42540 TO 42618 STEP 2: POKE dir,RND*200: POKE dir+1,RND*80:NEXT
```

For a detailed description of the command, see the "reference guide" chapter. In that chapter you will find different examples to simulate stars, earth, stars with two depth planes, rain or even snow. With imagination, it is probably possible to simulate more things with the same function. For example, if you place the stars in sequences of 2 or three pixels diagonally, instead of randomly distributing them, you can achieve a "segment" movement displacement, which could be ideal.

to simulate rain.



*Fig. 76 Rain effect with STARS*

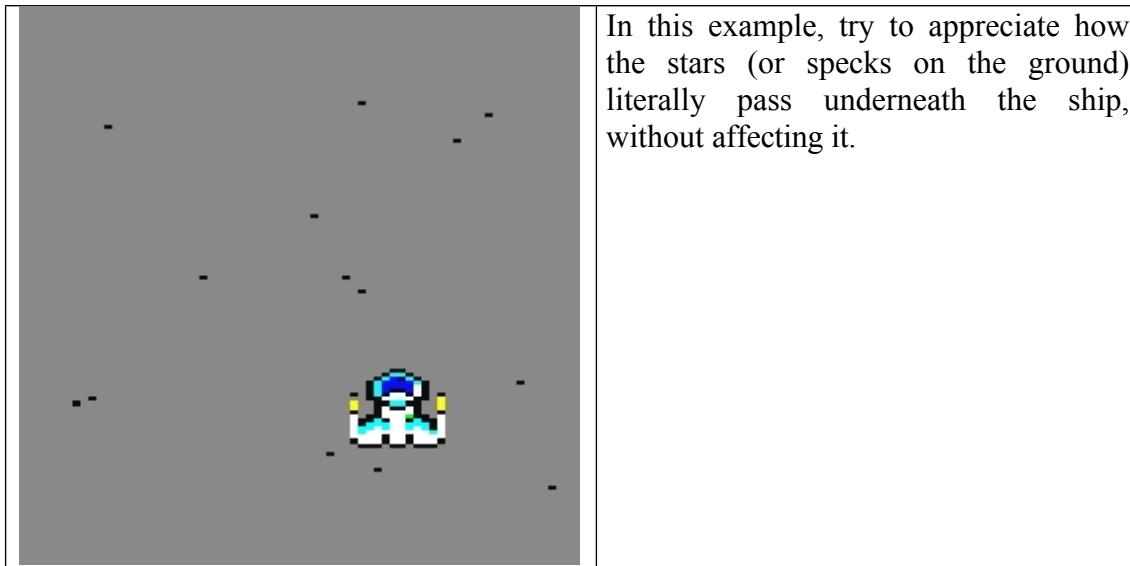
```

10 MEMORY 24999
20 CALL &6B78:' install RSX
40 mode 0:CALL &BC02:'restore default palette just in case'.
50 bank=42540
60 FOR dir=bank TO bank+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 'RAIN SCENARIO
141 '-----
150 |SETLIMITS,0,80,50,200: ' game screen limits
151 grass=&84d0:|SETUPSP,30,9,grass:'letter Y is sprite 31
152 rocks=&84f2:|SETUPSP,21,9,rocks: 'letter P is sprite 21
160 string$="YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY".
170 |LAYOUT,22,0,@cadena$:this paints the grass
180 string$="PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP"
190 |LAYOUT,23,0,@string$:paints a row of rocks
200 |LAYOUT,24,0,@string$:paints another row of rocks
210 '---- game cycle-----
211 defint a-z
220 LOCATE 1,10:PRINT "RAIN DEMO".
221 LOCATE 1,11:PRINT "press ENTER".
230 |STARS,0,40,4,4,2,-1
240 IF INKEY(18)=0 THEN 300
250 GOTO 230

```

As the example of a double plane of stars is in the reference chapter of the library, here is an example of a spacecraft flying over a speckled earth planet, with a vertical scrolling feel.



*Fig. 77 Mottled ground effect with STARS*

There is a way to invoke the STARS command in an optimised way and it consists of simply invoking it the first time with parameters and the following times without parameters. The command will assume that the values of the parameters are the same as those of the last invocation with parameters and this saves time that the BASIC interpreter spends processing the parameters, up to 1.7ms.

```

10 MEMORY 24999
11 'I put random stars
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:'restore default palette just in case'.
26 ink 0.13:'grey background
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: 'limits of the game screen

41 ' we are going to create a ship in sprite 31
42 |SETUPSP,31,0,&1:' status
43 ship = &a2f8: |SETUPSP,31,9,ship:' assign image to sprite 31
44 x=40:y=150: ' ship coordinates

49 '----- game cycle-----
50 |STARS,0,20,5,1,0:' black stars on a grey ground
55 gosub 100:' movement of the ship
60 |PRINTSPALL,0,0
70 goto 50

99 ' routine movement ship -----
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 using MOVERALL and/or AUTOALL

Now making combined use of relative movement, star scrolling and the order in which the sprites are printed, we are going to look at an example of how to simulate a spacecraft flying over a lunar landscape.

First of all, we have chosen sprite 31 for our ship, because that will make it print last. The sprites are printed in order, starting at zero and ending at 31. If a crater is a sprite lower than 31, it will be printed before the ship and the ship will be "above" it, giving the impression that it is flying over it.

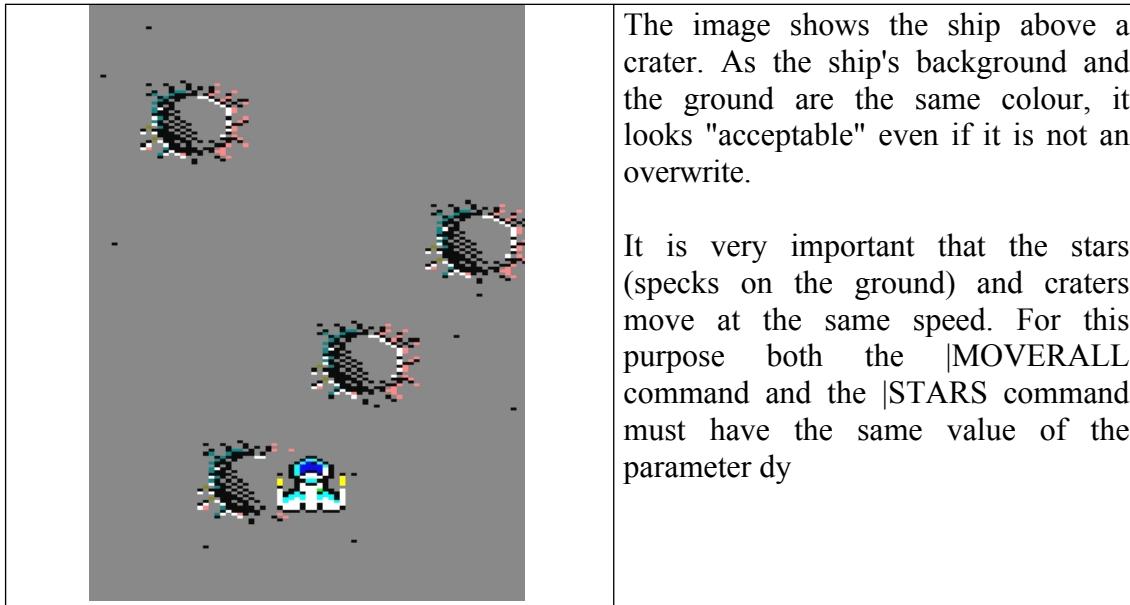


Fig. 78 flying over the moon

This is the BASIC code:

```
10 MEMORY 24999
11 'I put random stars
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:'restore default palette just in case'.
26 ink 0.13:'grey background
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' limits of the game screen

41 ' let's create a ship in the sprite 31
42 |SETUPSP,31,0,&1:' status
43 ship = &a2f8: |SETUPSP,31,9,ship:' assign image to sprite 31
45 x=40:y=150: ' ship coordinates

46 ' now the craters
47 crater=&a39a: cy%=0
48 for i=0 to 3 : |SETUPSP,i,9,crater:
49 |SETUPSP,i,0,&x10001: ' impression and relative motion
50 x(i)=rnd*40+20:y(i)=i*40
50 |locatesp,i,y(i),x(i)
70 next
```

The image shows the ship above a crater. As the ship's background and the ground are the same colour, it looks "acceptable" even if it is not an overwrite.

It is very important that the stars (specks on the ground) and craters move at the same speed. For this purpose both the |MOVERALL command and the |STARS command must have the same value of the parameter dy

```

71 t=0

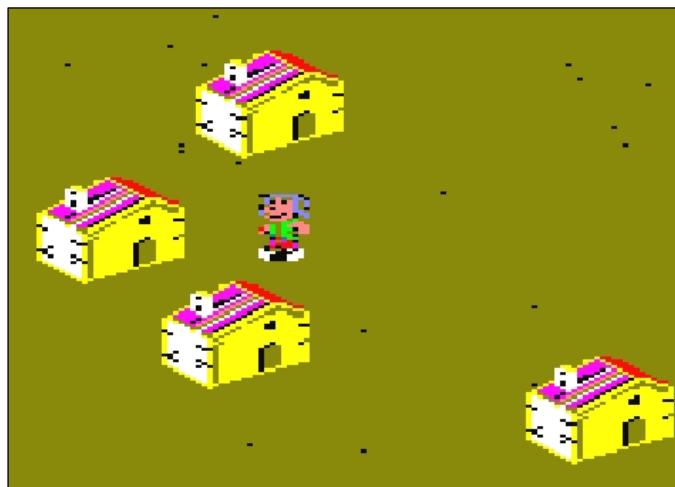
80 ----- game cycle-----
81 |STARS,0,20,5,3,0:' black stars movement
82 gosub 100:' movement of the ship
83 |MOVERALL,3,0: 'crater movement'.
84 t=t+1: if t> 10 then t=0:gosub 200:' crater control
90 |PRINTSPALL,0,0:' ship and crater printout
91 goto 81

99 ' routine movement ship -----
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 ' crater re-entry control
200 c=c+1: if c=6 then c=0
220 |PEEK,27001+c*16,@cy% 220 |PEEK,27001+c*16,@cy%
230 if cy%>200 then |POKE,27001+c*16,-20
240 return

```

Let's look at one last example that uses relative movement to give the sensation of scrolling, using sprites with drawings of houses, a mottled floor and a character located in the centre that, depending on the direction he is moving, makes everything move around him. It's a very basic example, but it gives you an idea of the potential of these functions - here it's the whole town that moves!



*Fig. 79 The whole village moves*

```

10 MEMORY 24999
20 MODE O: 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

```

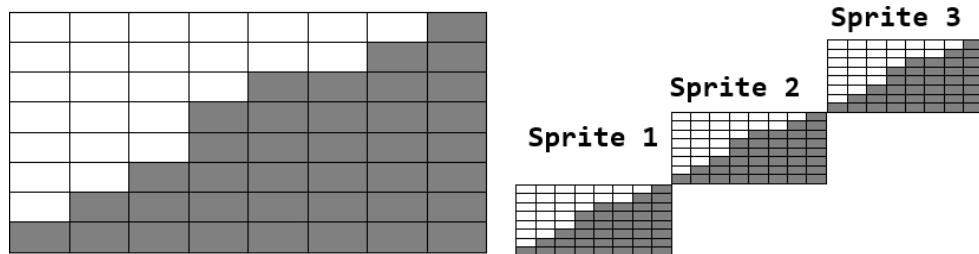
```

290 |SETUPSP,i,0,&X10001
300 |SETUPSP,i,9,&A01c:rem houses
301 |LOCATESP,i,RND*150+50,rnd*60+10
310 NEXT
320 |SETUPSP,31,7,6: rem character
330 |LOCATESP,31,90,38
340 |SETUPSP,31,0,0,&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,already,xa
480 GOTO 410

```

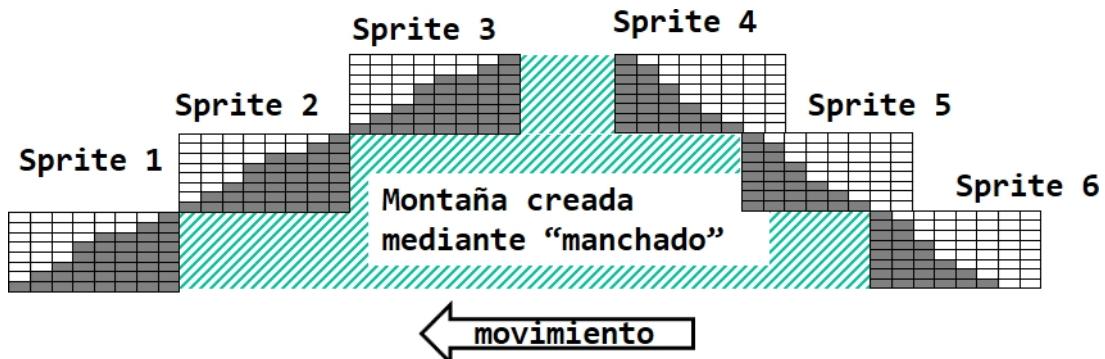
### 14.3 Technique of "spotting"

The technique for painting mountains in games with horizontal scrolling and lakes in games with vertical scrolling is the same. What we will do is to paint only the beginning of the mountain, using a sprite to paint its left side. We will put as many as we want. In this case I have put three



*Fig. 80. Defining the slope of a mountain with several sprites*

We do the same with a mirror image that we will associate with 3 other sprites, and we will place them on the right, building the right side of the mountain. Make sure that the mirror image has at least the last two columns of pixels at zero. This will allow it to erase itself as it moves to the left. Note that in 8BP sprites move byte by byte (two pixels at a time).

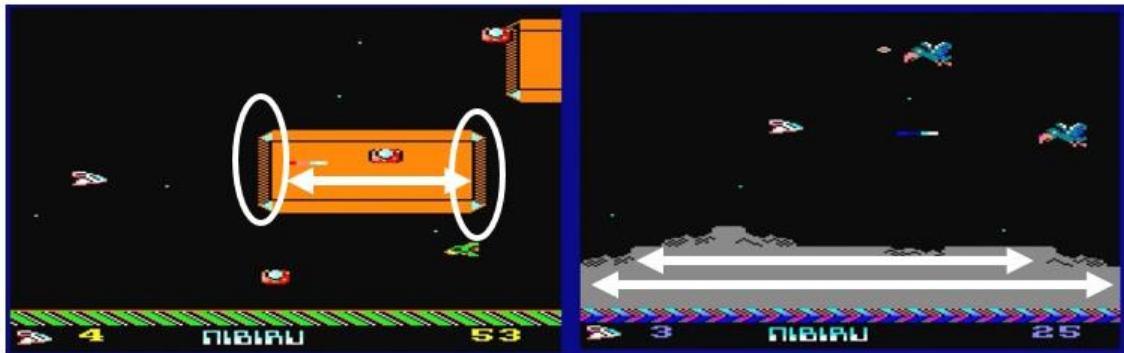


*Fig. 81 Mountain created with 6 sprites and the technique of "spotting".*

By moving all the sprites to the left by automatic or relative movement, the sprites on the left will start to "smudge" the background and therefore

"filling" the mountain, at the same time the sprites on the right will start cleaning it up. If the mountain gradually appears as it enters the screen, it will look like a huge mountain sprite, when in fact it is just 6 small sprites.

The video game "Nibiru" makes use of the "smudging" technique to draw the mountains and other large elements, in combination with the MAP2SP command that we will see later.



*Fig. 82. Examples of the staining technique*

I also used the smudging technique in the video game "Eridu", where huge mountains move smoothly across the screen.



*Fig. 83. Staining technique in "Eridu".*

In the case of a vertical scrolling game, if we want to paint a lake on a brown terrain, we will do the same, some sprites will "stain" the terrain and others further away will "clean" it, making it look like a huge lake, in one piece.

There is only one precaution you must take, and that is that the ships do not fly over the lake or your "trick" will be exposed! In case you want it to be possible to fly over the lake, then you must use sprite overwriting, as is done in stage 2 of the "Nibiru" videogame, where your ship and enemy ships can pass over large coloured rectangles without destroying them.



#### 14.4 MAP2SP: Scroll based on a map of the world

All the previous techniques are perfectly valid for scrolling, and even compatible with what we are going to see now, which is the fundamental technique that will allow you to design a "world map" and make your character or your ship scroll through it, with just one line of code.

To use the **|MAP2SP** command it is necessary to select the "assembly option" 2, which gives us 24.8 KB free for the BASIC listing.

The idea is simple: we will create a list of elements that make up the map of the world (up to 82 elements that we will call "map items"). Each element is described by the Y,X coordinates where it is located and the memory address where the image of the element in question is located (a house, a tree, etc.). The image associated with a map element can be any size you want. The coordinates of each element will be a positive integer, from 0 to 32000.

Once the map is created, we will invoke the function:

##### **|MAP2SP, Yo, Xo**

This function analyses the list of items in the world and determines which of them are being displayed if the world is viewed by placing the bottom corner of the screen at the coordinates (I, Xo). The function transforms the "map items" into sprites, occupying the positions in the sprite table from zero onwards. This may consume many or few sprites, depending on the density of map items you have. On a subsequent invocation of the same function, map items that are no longer present in the scene will not consume sprites in the table, and other map items will take over. This means that **the MAP2SP function consumes a variable and indeterminate number of sprites, depending on the number of map items visible on the screen at any given time**. In the example below you would use 3 sprites when invoking MAP2SP at the coordinates indicated.

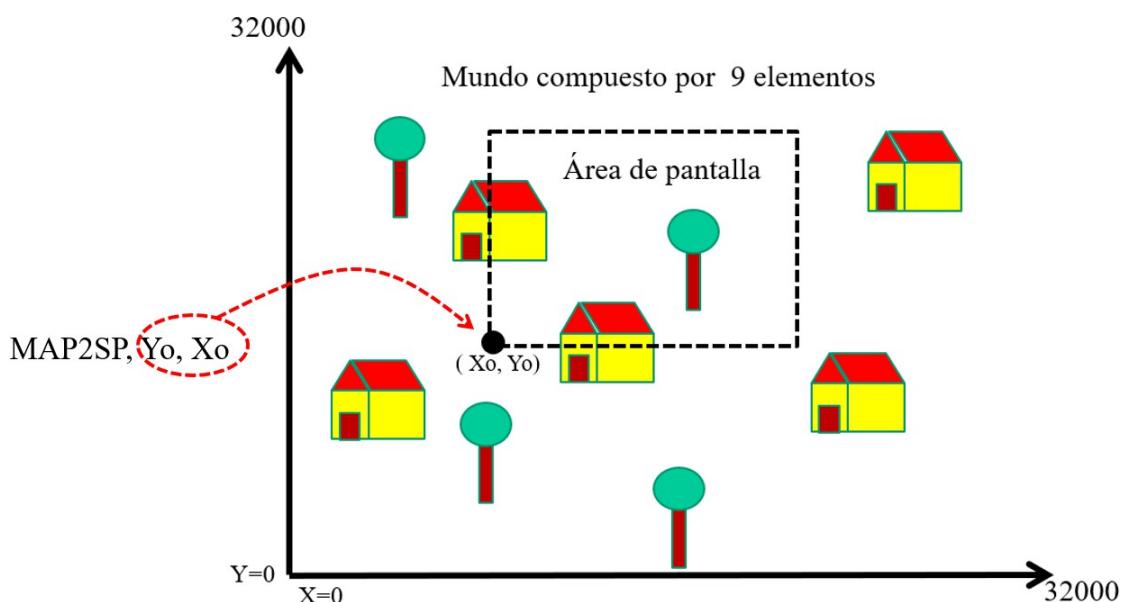
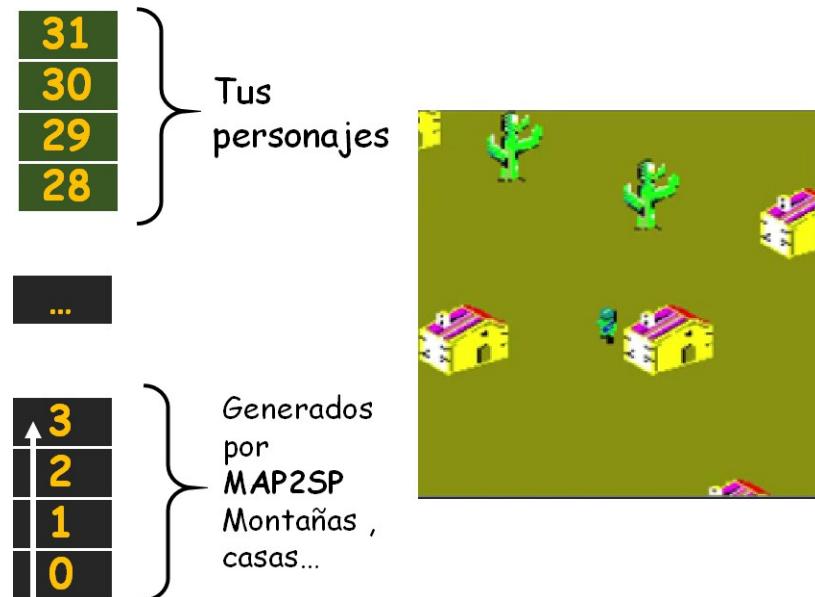


Fig. 84 Map of the world and MAP2SP

If you use this mechanism, your character and enemies must use sprites from 31 downwards, thus avoiding possible clashes between the sprites used by the scroll mechanism and your characters. If by any chance MAP2SP encounters more than 32 items to translate to sprites, it will ignore those exceeding 32.



*Fig. 85 sprites consumed by MAP2SP*

You must invoke MAP2SP every game cycle or at least every time you change the coordinates of the viewpoint from which you want to view the world. The "sliding" window with which you hunt the images that are transformed into sprites always measures what the screen measures (80 bytes x 200 lines) regardless of how you set the SETLIMITS command. That is, **even if SETLIMITS sets a very small painting area, whatever the 80 bytes x 200 lines window has "hunted" will be transformed into sprites**.

The sprites created by MAP2SP are created by default with state 3, i.e. with the print flag active (PRINTSPALL prints it) and with the collision flag active (COLSP will collide with it). If you need the sprites to be created with another state, you simply invoke the MAP2SP command once with a single parameter indicating the state with which the sprites should be created. With a single invocation of this type, the command is configured for the following invocations with coordinates.

|MAP2SP, <status>, <status>, <status>, <status>, <status>, <status>, <status>.

Example:

|**This configures the MAP2SP command to be printed but not collidable.**

Having clarified the concept let's review in detail how the world map is specified and an example of using the MAP2SP function.

#### 14.4.1 Map of the World (Map Table)

The table where we will register all the elements of the map is called MAP\_TABLE and is specified in an .asm file called **map\_table\_tujuego.asm**

This table contains the items that define the world map images for your scrolling games. The table is assembled at the same memory address as the LAYOUT, i.e. address 42040. This means that the layout and the world map cannot be used simultaneously, but this is not a problem since a scrolling game will not use the layout and vice versa. Also, the restriction is only that they cannot be used at the same time, but a game could have one phase where it uses the layout and another where it scrolls based on the world map.

The world map input table starts with 3 global parameters (occupying 5 bytes in total) and a list of "map items", which are described by 3 parameters each (x, y, image direction).

The list can contain up to 82 items, but the number of items can be limited by one of the global parameters. The list, at most, occupies the initial 5 bytes + 82 items x 6 bytes =  $5+492=497$  bytes. If we put one more item, we would exceed the 500Bytes reserved for the map (which are the same reserved for the Layout).

The table starts with 3 parameters:

- The maximum height of any map item
- Maximum width of any map item (to be expressed as a negative number)
- Number of items (maximum 82)

The first two parameters are important to check when a sprite may be partially appearing on screen, as the MAP2SP function does not know or find out the width and height of each image. It only knows where the map item is located and assuming the maximum width and height, it checks if that item may be entering the screen. If so, a sprite is created from the map item. If these two parameters are set to zero, the top left corner of the map item needs to be inside the screen for that item to be transformed into a sprite.

Each item is a tuple of 3 parameters preceded by the mnemonic "DW":

**DW Y, X, <image>**

Let's see an example of the file called **map\_table\_tujuego.asm**

```
MAP TABLE
;-----
3 parameters before the list of "map items".
dw 50; maximum height of a sprite in case it gets through the top and you have to paint part of it.
dw -40; maximum width of a sprite in case of a left-hand sprite (negative number)
db 82; number of map elements.at most it should be 82

and from here start the items dw
100,10,HOUSE; 1
dw 50,-10,CACTUS;2
dw 210,0,HOME;3
```

```

dw 200,20,CACTUS;4
dw 100,40,HOUSE;5
dw 160,60,HOUSE;6
dw 70,70,HOUSE;7
dw 175,40,CACTUS;8
dw 10,50,HOUSE;9
dw 250,50,HOUSE;10
dw 260,70,HOUSE;11
dw 260,70,HOUSE;11
dw 290,60,CACTUS;12
dw 180,90,HOUSE;13
dw 60,100,HOUSE;14
dw 60,100,HOUSE;14

```

...

To design your world, I recommend that you take a checkered notebook and draw on it the elements you want your world to have. Each square of the notebook can represent a fixed amount such as 8 pixels or 25 pixels. The point is that you should take your time to draw the world you want and the way you want to go through it. For example, there are multi-directional "Gauntlet" type games and other vertical scrolling games like Commando. It's your choice, but in any case do it with time and patience and the result will be worth it.

Each phase of your game can be a map. In 8BP you can change the map whenever you want using POKE functions. I normally use 1KB out of the memory space used by 8BP, for example from 23000 to 24000, to store all the phases (maps) of the game and every time I enter a phase I load at address 42040 the corresponding map by PEEKing and POKEing. That is, I build my map file at address 23000 and it occupies 1Kbyte, leaving 23 KB for my BASIC program. In order that this map information is not crushed by basic, I will have to make a MEMORY 22999 at the beginning of the game.

#### **14.4.2 Using the MAP2SP function**

Now let's see an example of how to use this function. Basically it has to be invoked once in each game cycle with the new coordinates of the origin from where the world is observed.

The function will create a variable number of sprites from sprite 0 onwards and will create them with their screen coordinates adapted. That is to say, even if a map item has a coordinate x=100, if the moving origin is located at position x=90 then that sprite will be created with the screen coordinate x'=x-90=10. The Y-axis coordinate will take into account that the Y-axis on the Amstrad grows downwards, while the world map grows upwards. So the Y-coordinate is adapted using the equation  $Y' = 200 - (Y - Y_{orig})$ . But don't worry, this adaptation is already done by the MAP2SP function. You just have to change the moving origin from where the world map should be displayed.

In this mini-game, a world of houses and cacti has been created and our character walks between the elements. In this example, in the event of a collision (detected with **|COLSPALL**), the character will not be able to continue. In an aircraft game where map items are "flyable", we could parameterise the collision to only detect collisions with enemies and gunfire and not with background elements, using **|COLSP, 32, <sprite\_initial>, <sprite\_final>**.



*Fig. 86 Mini scrolling game inspired by "commando".*

As you can see, it's very small, but it has everything: multidirectional scrolling, keyboard reading, changing character animation sequences, collision detection, music...

**IMPORTANT: note the MEMORY command. We have used assembly option 2, ideal for scrolling games which leaves us almost 25 KB free.**

```

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:' initial direction upwards
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, 0: REM collision as soon as there is a minimum overlap
190 'begins game cycle
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

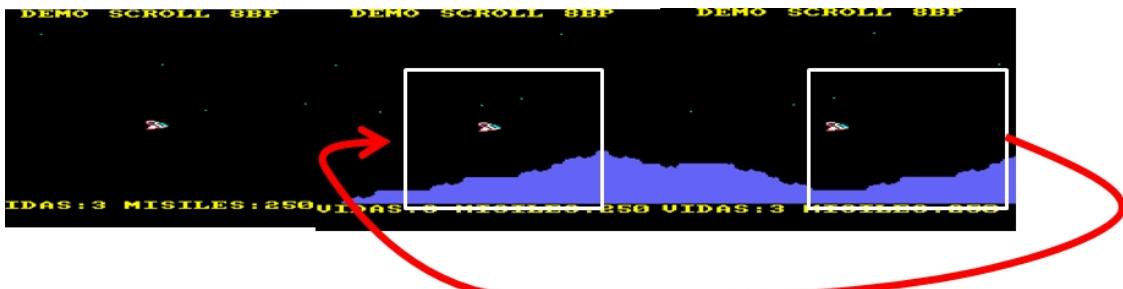
```

```

270 GOTO 200
280 |MUSIC:MODE 1: INK 0,0: PEN 1

```

Let's now look at another example of horizontal scrolling, where an interesting effect of an "infinite" world map has been achieved, making the end of the map equal to the beginning and causing an abrupt jump when Xo reaches a certain value. In fact, this world map has only 13 elements.



*Fig. 87 Map of the "infinite" world*

This is the map that has been used

```

_MAP_TABLE
3 parameters before the list of "map items".
dw 50; maximum height of a sprite in case it gets through the top and
part of it has to be painted.
dw -18; maximum width of any map item. must be expressed as a negative
number.
db 13; number of map elements. at most it should be 82

and from here start the 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
dw 48,190,MONTDW;8
here I repeat elements to fit with position 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
dw 60,270,MONTH;13
;
```

And this is the BASIC programme, where I have highlighted the line where the world goes backwards without the player noticing anything.

```

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,0,0,0,5
110 xo=0:yo=0
111 x=36:y=100
120 |SETUPSP,31,0,&X100001
130 |SETUPSP,31,7,1:dir=1: initial direction upwards
140 |LOCATESP,31,y,x
160 |SETLIMITS,0,80,0,176: |PRINTSPALL,0,1,0
161 LOCATE 1,23 :PEN 1: PRINT "LIVES:3 MISSILES:250" PRINT "LIVES:3
MISSILES:250" PRINT "LIVES:3 MISSILES:250" PRINT "LIVES:3 MISSILES:250
162 LOCATE 1,1:PRINT " DEMO SCROLL 8BP"
170 col%=32:sp%=32:|COLSPALL,@sp%,@col%
180 |COLSP, 34, 0, 0, 0: REM collision as soon as there is a minimum overlap
190 'begins game cycle
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 END
260 |PRINTSPALL
261 cycle=cycle +1: IF cycle=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 Example of a phase file

If you want to have several stages in a scrolling game, as I said before, you can have them preloaded in a memory area. For example, you can assemble the stages at address 23000 and then you have 1000 bytes to store several world maps, since 8BP starts at address 24000. In that case your game will have to start with a MEMORY 22999.

The "Nibiru" videogame does this, although it was created when 8BP started at address 26000 (it was created with 8BP v26), so it stores the map from 25000 onwards. To load a phase it simply reads the area where each phase has been assembled and writes it over the address where the world table should be before starting to play in that phase. These three BASIC lines show how to copy phase 1 of the game (&61a8 = 25000)

```

310 ' pokes from the world map
320 dirmap!=42040:FOR i!=&61A8 TO &620D
330 dato=PEEK(i!):POKE dirmap!,dato:dirmap!=dirmap!+1
340 NEXT

```

There is a faster way to load the phases, using the |UMAP command which I will explain later, but this FOR loop with POKEs is perfectly valid.

Finally, next, I show you the stages file of the game "**Nibiru**", which we assemble at address 25000 (remember that the version of 8bp with which "**Nibiru**" was created started at 26000, so from 25000 to 26000 there were 1000 bytes free. With the current version of 8BP we would assemble the phases without reaching address 24800).

```
org 25000
PHASE1
;=====
_START_PHASE1
3 parameters before the list of "map items".
dw 50; maximum height of a sprite in case it gets through the top and part of it has to be painted.
dw -18; maximum width of any map item. must be expressed as a negative number.
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 dw
60,172,MONTH;6 dw
60,194,MONTDW;7 dw
48,206,MONTDW;8
here I repeat elements to fit with position 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
dw 36,380,MONTDW;16
_END_PHASE1
;=====
PHASE2
;=====
_START_PHASE2
dw 50; maximum height of a sprite in case it gets through the top and part of it has to be painted.
dw -6; maximum width of any map item. must be expressed as negative number
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,PLACA2_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,PLACA2_R_OV dw
156,254,CUPULA2_OV
```

```

-END_PHASE2
=====
PHASE3
=====
_START_PHASE3
dw 50; maximum height of a sprite in case it gets through the top and you have
to paint part of it.
dw -80; maximum width of any map item. must be expressed as a negative number.
db 4
dw 40,0,MAR; 1
dw 40,80,SEA; 2
dw 189,0,CLOUDS; 2
dw 189,80,CLOUDS; 2
-END_PHASE3

```

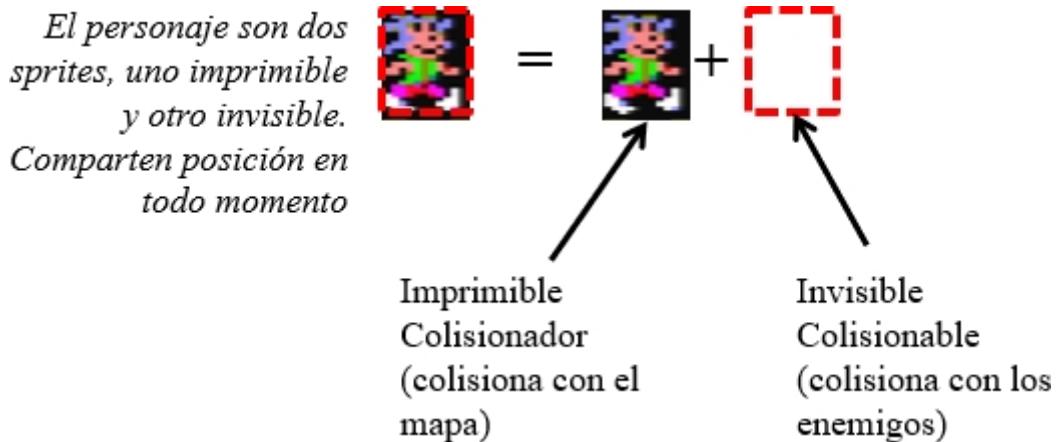
#### 14.4.4      Enemy collision with map

You may want to make a game where your character collides with the map (using COLSPALL) and is therefore a collider. Let's say you have this:

- Your character: collider
- Map elements: colliderable
- Your shot: collider
- Enemies: colliderable

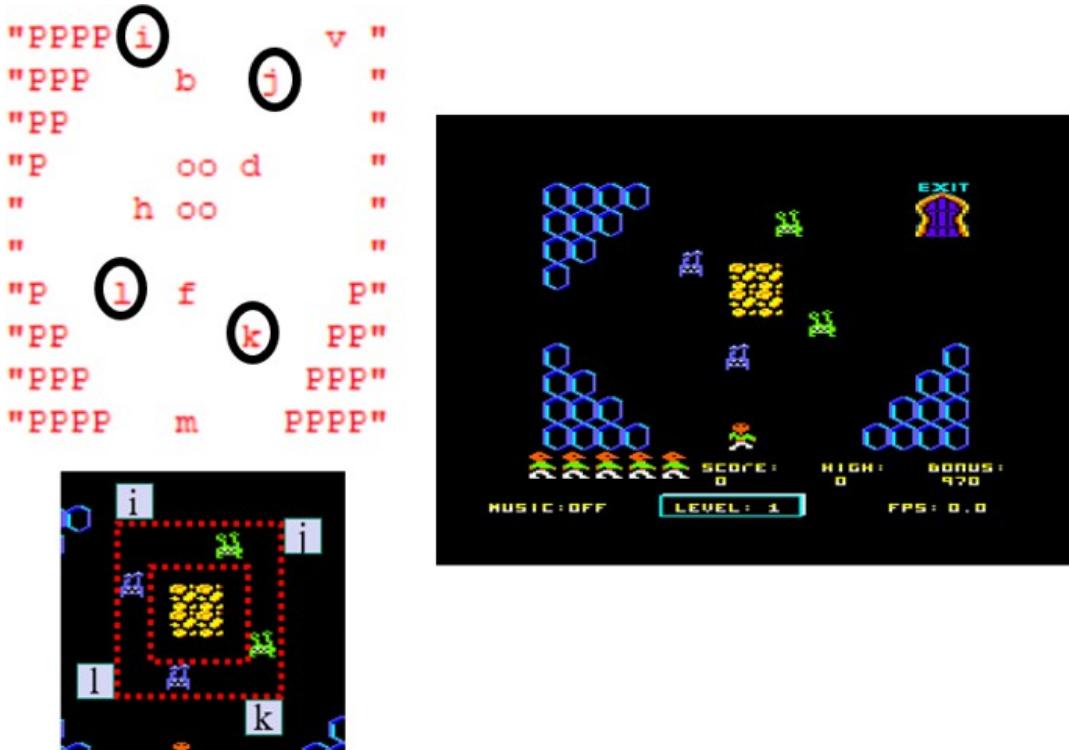
Now, if your enemies are colliders, then they can't collide with the map, and maybe you're interested in making a gauntlet game, for example. The solution is to make the enemies colliders. But of course, in that case they can't kill you anymore. And there's also a problem with the shot being collider as well.

The solution to this problem is based on using a simple trick based on non-printable "invisible" sprites, which don't spend CPU to print but are there. **Your character will spend two sprites**: a printable collider sprite and a non-printable collider sprite, but of the same size. With the shots we do the same, so they can collide with enemies and the map.



A similar trick is used in the game "**Eternal Frogger**". By using invisible sprites, the frog is killed by falling into the river (see the section explaining the COLSPALL command). This simple trick can also be applied to the shots of the

character. Invisible" sprites are very useful. In the game "Happy Monty" they are used to make enemies change direction, so that when an enemy collides with an invisible sprite, it changes its direction. This meant that it was not necessary to define a lot of routes adapted to each screen, but only to place in each level a series of invisible sprites that allowed to alter the trajectories of the enemies (see "making off" document of Happy Monty).



#### 14.4.5 Background images in your scroll

Background images are intended for scrolling games and are a feature provided by 8BP from version V42 onwards. It is a type of transparent printing in which the sprites that make up the background (the world map) can be printed underneath your character and enemies without causing flickering.

Background images whenever they are assigned to a Sprite are printed with this special transparency and it doesn't matter if the Sprite has the transparency flag assigned to its status byte or not. See chapter 8 to learn how to use them.

**IMPORTANT:** On the world map you can combine normal images with "background images" (section 8.5). Background images always have transparency. The transparency flag you use in **|MAP2SP, <status>** will only apply to normal images.

**IMPORTANT:** background images are expensive to print, and if you flip them, they are even more expensive. If your game has scroll, try to use them for elements that your character or enemies will fly over. For example, to speed up scrolling use

normal images on houses or rocks appearing on the sides of your scroll, which will not often be overlapped by the sprites

## 14.5 Parallax Scroll

Let's see how to make a parallax scroll, i.e. with several "planes" at different speeds. Maybe you can think of another way to do it, this is just one possible way, used in the game "Nibiru".

The first thing you should know is that it is much faster to print one very long horizontal sprite than many small sprites. This is because of the operations that have to be performed after finishing painting each scanline of a sprite. For the same reason it is much faster to print a very large horizontal sprite than a vertical sprite of the same size.

We will place two giant sprites on the world map to make the water. I made one 160 x8 so I put two of them so that when it scrolls, the next one starts to appear. When the MAP2SP function goes all the way around the screen, it goes back to x=0 and the whole thing repeats indefinitely. On the world map I also put two sprites for the clouds.

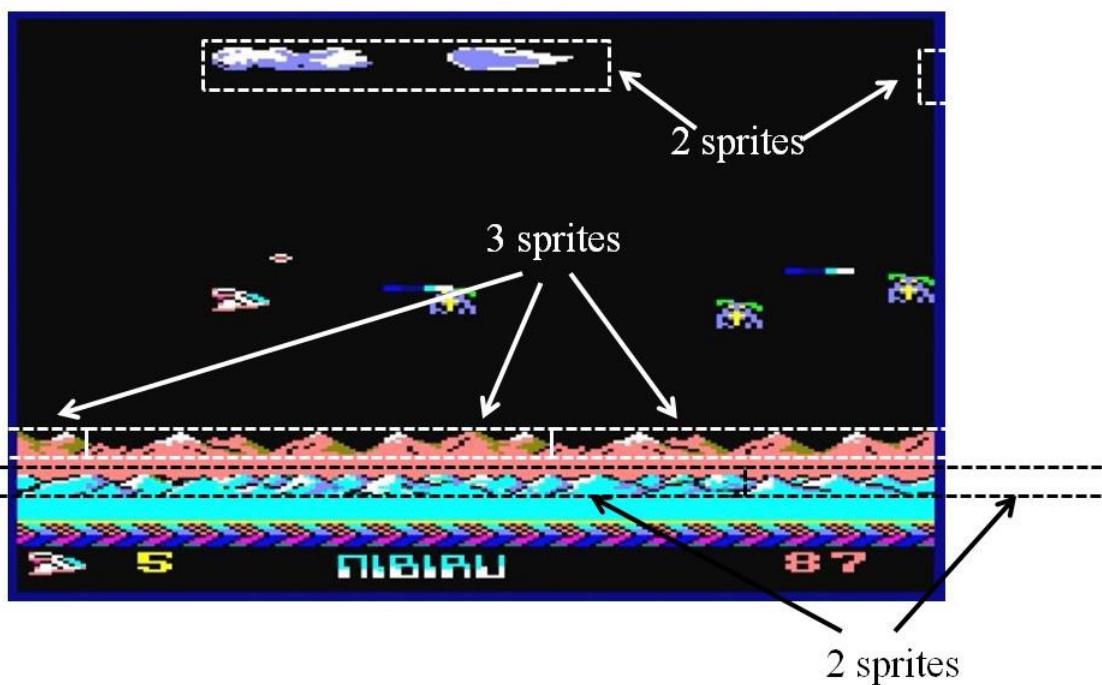


Fig. 88 parallax scroll

For the mountains I used 3 normal sprites, outside the world map. I gave them automatic movement on their status flag and made them move to the left. But on odd cycles I disable both the print flag and the automatic movement flag, so they only move and print every other game cycle, achieving a speed half that of the water. The sprites of the mountains are 16,17 and 18 and with these pokes I act on their status byte.

```
mc=cycle AND 1: IF mc THEN POKE 27256,0: POKE 27272,0: POKE 27288,0  
ELSE POKE 27256,11: POKE 27272,11: POKE 27288,11
```

In the example "mc" is the variable that determines whether the cycle is odd or even.

## **14.6 Dynamic map update: |UMAP**

Maybe in our game we need a map with more than 82 elements. Or we simply want the |MAP2SP command to run faster using a smaller map that we update periodically. Or we want both!

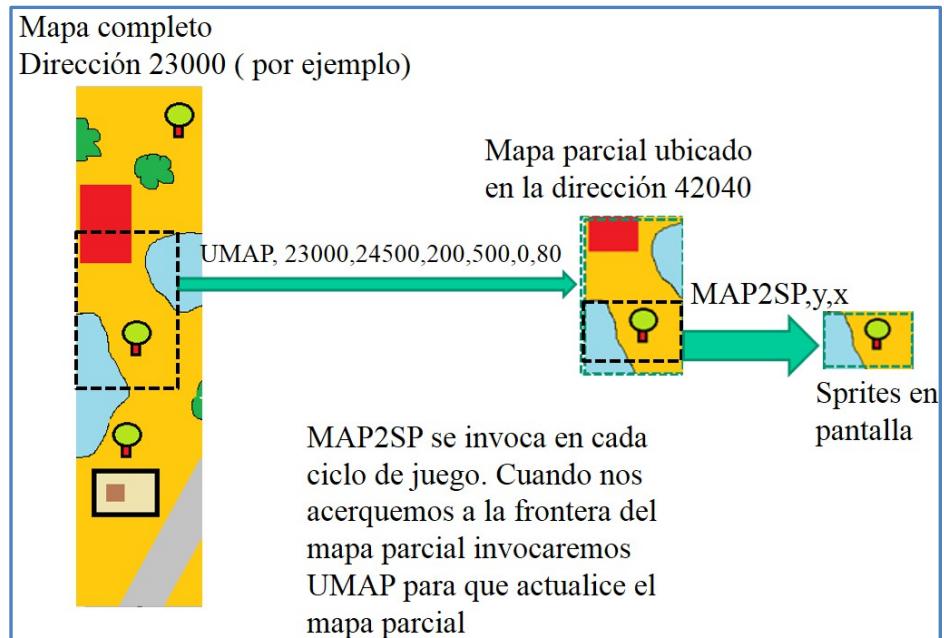
For this purpose, since version 32 of 8BP, there is the command **|UMAP** (short for "UPDATE MAP"). This command updates the map with information located in another memory area where we have a larger map. The command causes the map to be completely rebuilt, including only those items that meet certain ranges of X, Y coordinates (all parameters are 16-bit numbers).

```
|UMAP, <map_ini>, <map_fin>, <y_ini>, <y_fin>, <x_ini>, <x_fin>,  
<x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>.
```

UMAP is not a simple copying of elements. It is a "selective" copying. For example, if we have a map located at address 22500 that occupies 1500bytes and we want to update the map with the coordinates of our character, with enough margin to advance in the Y-coordinate up to 100 lines and in the x-coordinate up to 20 bytes in all directions:

```
|UMAP, 22500,23999, y-100, y+100, x-20, x+20
```

This command will check the coordinates of the items located on the map at address 22500 and if they are within the X, Y margins that we have set, they will be copied to the memory area that 8BP uses for the |MAP2SP command, that is to say, it will copy them from address 42040. However, it will only copy those that meet the condition. As there are fewer items, the |MAP2SP command will run faster as it will have to read and check if there are fewer items on the screen.



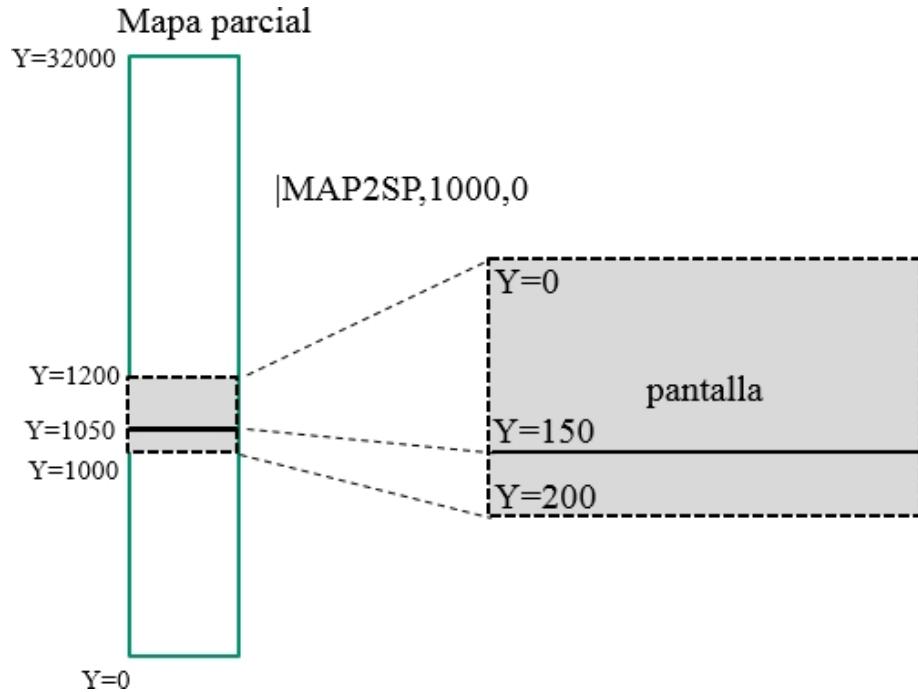
**Fig. 89 UMAP**

Periodically (not every game cycle) we can update the map with UMAP and thus we can create very large worlds with many elements and at the same time we will get more speed in MAP2SP. The UMAP command is very fast, but invoking it every game cycle does not make sense as MAP2SP can work with a much larger map than what fits on the screen and we can invoke UMAP only when we need areas of the original map that are not present in the partial map. I have used it in the car racing game "**3D Racing one**", as the track map was very large (exceeding 82 elements) and what I did was to invoke |UMAP periodically as the car progresses.

In the full map address (in the example 22500), we will only have a list of items. **We will not have the 3 initial data** contained in the 42040 map (I mean the maximum height of any map item, the maximum width of any map item and the number of items). The number of items is updated by |UMAP (depending on how many items meet the imposed margins). The other two parameters are set by you in the "map\_table\_your\_game.asm" file.

The |UMAP command puts the items into the partial map in an order intended to sort the partial map according to the Y coordinate of the screen. Normally you edit your global map in a file called "misupermapa.asm" or something like that. And assemble it in the 22500 (for example). In this file you will write one by one the items of your map, in ascending order of Y-coordinate. Well, to get the sprites already sorted by Y-coordinate (in ascending screen coordinate order), the |UMAP command reads them from the end to the beginning. That way the sprites generated later with |MAP2SP will be sorted by screen Y-coordinate. Remember that the screen uses an inverse coordinate system with respect to the map, that is, the 150th coordinate of the screen is the 50th coordinate of the map (in the case of |MAP2SP,0,0). If you don't understand this very well, don't worry. It is something of the inner workings of |UMAP and |MAP2SP just to make it more efficient. In the following figure I have represented the map and the CPC screen. As you can see

the map can be very large but also its coordinates grow upwards while the screen coordinates grow downwards.



*Fig. 90 MAP and Display*

#### **14.7 Animation and ink scrolling: RINK command**

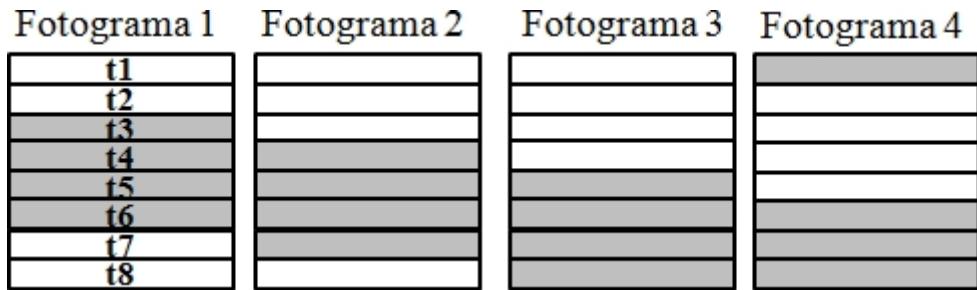
There are games that require moving large blocks of screen memory to give a sense of movement, such as the sidebars in racing games or large blocks of bricks or dirt. The MAP2SP command allows you to do all that, but the speed is not blazing fast because it spends a lot of CPU moving sprites. Dye animation is the perfect complement in these cases.

On computers like the AMSTRAD with its powerful palette of 16 simultaneous colours, many games make use of ink animation. A clear example are some car games whose roadside stripes lend themselves to this type of animation.



*Fig. 91 Ink-animated stripes*

Animation by inks consists of defining a set of inks on which a set of colours will be rotated. Let's see an example with white/grey stripes and 8 inks:



*Fig. 92 Ink animation*

Basically, to give a sense of movement, the first thing to do is to assign the colours to the inks that are going to rotate. In this case the colours white (26) and grey (26) and grey (26).

(13) are assigned to inks t1..t8. Let's assume that ink t1 is 8, so ink t8 will be 15. The rest of the inks (0 to 7) will be used for the sprites. At each time instant we will have to reassign the values of the 8 inks to give the sensation of rotation. This is what the |RINK command (short for "Rotate INK") does.

RINK allows you to define a pattern of colours to rotate over a set of inks. An ink is not a colour. An ink is an identifier in the range [0..15] that identifies a colour in the range [0..26]. To define the pattern of colours to rotate, we will use the RINK command as follows:

**RINK, <initial\_ink>, <colour1>,<colour2>, <colour3>, ... ,<colourN>, <colourN>, ... ,<colourN>, <colourN>, <colourN>.**

This indicates that they will rotate N inks starting with the initial ink using the colour pattern indicated. Note that if you use a lot of inks to make an animation, you will have fewer inks left for your sprites.

Once the pattern has been established, we can rotate the inks more or less quickly with

**RINK, <step>**

The step value is the number of displacements that the inks in the stencil will undergo and therefore a higher value gives a higher speed displacement effect.

Recommendation: Due to the use of RINK interrupts, RINK sometimes "stalls" when used at the same time as the |MUSIC command at speed 6. In case you want to use both at the same time without interference, use another speed for music (you can use speed 5 or 7, both will work fine).

### 14.7.1 2D car racing

The example of the car game can be found in the following list. The sound of the engine is intended to give the feeling of acceleration. For the signs on the sides, a relative motion sprite has been used, which moves at the same speed as the step of the stripes. The ink pattern is defined in line 100.

## | RINK,1,3,3,3,3,3,24,24,24,24: rem yellow and red stripes

```
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 pallet
50 GOSUB 430
60 INK 0,13
70 INK 14,10
80 linestinta=3
90 rangotintas=8
91 ' establishment of the ink pattern
100 |RINK,1,3,3,3,3,3,24,24,24,24,24: rem yellow and red stripes
101 |RINK,0
110 y=400
120 ' PAINT ROAD -----
121 tini=1
130 FOR strips=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 strips
250 skipb=-16:xc=65: cosa=0
270 REM play cycle -----
293 IF saltob=-16 THEN 296
294 IF jumpb>0 THEN thing=-jump ELSE thing=thing-1
295 IF thing<0 THEN |RINK,thing:vy=3*thing:posv=posv-3*thing:|MOVERALL,-vy,0:IF
saltob <=0 THEN thing=2-saltob
296 |PRINTSPALL
351 cycle=cycle+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 jumpb<16 THEN jumpb=jumpb+1:jump=jumpb/4
380 IF INKEY(69)=0 THEN IF jumpb>-16 THEN jumpb=jumpb-1:jump=jumpb/4
390 SOUND 1 ,6000/(skip+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 Brick Scroll

In this example we are going to combine the use of ink animation with **|MAP2SP** based scrolling. Using ink animation we will move a pattern of bricks that we have drawn with a repeating sprite while the castle and the tree will be moved by **|MAP2SP**.

Moving the bricks would be a huge job if done by the CPU, so this technique allows the "impossible". It is a very powerful technique if you use it with ingenuity.



*Fig. 93 Scroll using ink animation and MAP2SP at the same time*

The brick used is actually an 8-colour sprite, with the design shown below:

								0
8	9	10	11	12	13	14	15	

And the ink pattern is 0,6,3,3,3,3,3,3,3,3,3,3 . to create it you simply run the command:  
|RINK,8,0,6,3,3,3,3,3,3,3,3,3

The character (sprite 31) remains in the centre of the screen, being able to jump and move in both directions. To synchronise the movement of inks and MAP2SP, a step=2 is set for the |RINK command, as one byte is two pixels and MAP2SP moves at the byte level.

It is interesting to note how the bird (sprite 30) must also be affected by the movement as its speed must be added to or subtracted from that of the character.

As for the colour, since an 8-colour pattern is used and overwriting is also used, we are left with 2 colours for the background (castle, branches) and 3 colours for the sprites (character and bird). It is easy to modify the program to use a 4-colour rotation pattern and thus have 5 colours for sprites, which is more reasonable.

The full list is below.

```
10 MEMORY 24999
11 ON BREAK GOSUB 5000
20 CALL &6B78
30 FOR i=0 TO 31:|SETUPSP,i,0,0,0:NEXT:|AUTOALL,1:|PRINTSPALL,0,1,0
40 |SETUPSP,31,7,1:|SETUPSP,31,0,65:|LOCATESP,31,130,36:'character
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 palette
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 ' establishment of the ink pattern
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 and |MAP2SP".
220 ' PAINT wall -----
230 |SETUPSP,29,9,23:'brick
231 y=152
232 FOR row=1 TO 6
240 FOR brick=xini to 42 step 2:|PRINTSP,29,y,brick*2:next
241 xini=(xini-1) mod 2: y=y+8
242 next
390 dir=1:x=0:xp=80: cycle=40: stepy=2
400 |MUSIC,0,0,0,7
409 ' game cycle -----
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=cycle:|SETUPSP,31,0,205:|SETUPSP,31,15,dir-1:|SETUPSP,31,7,31+dir
472 IF cycle-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:'bird start again
501 cycle=cycle+1
502 IF xant=x THEN |MAP2SP,0,32000
503 xant=x:' IF still THEN I don't print the castle so it doesn't flash
510 GOTO 410
5000 |MUSIC:CALL &BC02:pen 1:MODE 2:END
```

## 15 Platform games

The difficulty of programming a platform game lies mainly in correctly handling the physics of jumps and platform collisions. Something you can find in the video game: "Fresh fruits & vegetables" available at 8BP.



Fig. 94 Fresh fruits & vegetables

### Jumps:

Basically for the physics of the jumps instead of using Newton's equation I have defined a path where Vy starts at -5 and decreases frame by frame. When the zenith position is reached, the image is changed so that it erases itself at the top and the velocity Vy becomes positive, and gradually increases.

In essence, it is like applying Newton's equation, but without the calculations.

### Soil check

While the character is walking, I check each frame to see if there is a floor. As the platforms belong to the world map, they use low sprite identifiers (<10) so if with COLSPALL I detect a number <10, there is ground. If the result of the detection is a 32 (sprites go from 0 to 31) then there is nothing and the character must start falling. Enemies don't detect anything. They simply walk along predefined routes, where it is indicated how many steps they must take in each direction and then start again. From then on, the sprite runs the route step by step by invoking |AUTOALL (which internally already invokes |ROUTEALL).

### Platform collisions:

when the character is on the fall path, I move it (without printing) down 5 units and detect sprite collision. Then, I move it (without printing) 5 units upwards. If the collision is 32, there is no collision and I keep the character falling. If the collision is less than 10, the character has collided with a platform. In that case I move the character so that it fits perfectly with the beginning of the platform, so: position of the doll's head is "posy" position of the feet is posy+26 because the character measures 21 and I add 5 because it is falling (there is 5 of self-deletion), so that in reality it measures 26. As the platforms have been placed in multiples of 8, I simply keep the rest of the whole division, which I can obtain with an AND 7 in short, two very well thought out instructions, but only two:

```
dy =(posy%+26) AND 7  
|MOVER,31,5-dy,0
```

Then I assign the walking animation sequence, which is not the falling sequence and its frames are 21 frames high.



## 16 Hordes of enemies in scrolling games

Scrolling games are usually played as a succession of hordes of enemies of different types and trajectories, as you advance vertically or horizontally.

If you wanted your map to have 10 hordes at different instants of map progress (or game cycle) you could use 10 IF statements, but it would be very slow to execute each cycle (each IF check costs one millisecond).

The best solution is to maintain two arrays, one for horde position and one for horde type.

Index (Sequential order number)	Nexthorda (Cycle in which it should appear)	Tipohorda (Type of horde enemies)
0	100	1 - aircraft
1	200	2 - missiles
2	250	4 - UFOs
3	320	3 - dragons

```
10 dim tipohorda(10)
20 typeforda(0)=1: typeforda(1)=2: typeforda(2)=4:
t y p e f o r d a (3)=3:
30 dim nexthorda(0)
40 nexthorda(0)=100:nexthorda(1)=200: nexthorda(2)=250 ...
50 index=0

100 rem game cycle
110 cycle=cycle +1
120 IF cycle = nexthorda(index) THEN GOSUB 500
...
500 rem routine creation horde
510 on tipohorda(index) goto 600,700,800
520 rem routin creation horde type 4. Al final we will goto 1000
      do
600 rem routin creation horde type 1. Al final we will goto 1000
      do
700 rem routin creation horde type 2. Al final we will goto 1000
      do
800 rem routin creation horde type 3. Al final we will goto 1000
1000 Index=e
1010 RETURN
```

Instead of using the cycle in which it should appear you can also make the comparison with the map position you are in, that will depend on how you want to do it, but with this idea of the two arrays you already have the general approach to organise your hordes of enemies.



## 17 Redefinable mini-characters: PRINTAT

The Amstrad character set is nice and well constructed. However, in mode 0 we only have 20 characters wide per line and they appear too "wide", so sometimes they are not suitable for displaying certain texts or markers in a game. Also, the PRINT command is very slow, so it is not recommended to update the markers frequently, as the game "stops" while it is printing, it is only a few milliseconds, but it is noticeable.

For that reason, from v31 version of 8BP the PRINTAT command has been added, which can print a string of characters using a new, smaller set of characters (I call them "mini-characters"). This new command allows you to use the transparency mechanism of the sprites, so you can print characters respecting the background. It works as follows:

**|PRINTAT, <flag transparency>, y, x, @string**

Example:

**cad\$= "Hello".**

**|PRINTAT, 0, 100, 10, @cad\$**



*Fig. 95 PRINTAT*

The |PRINTAT command prints character strings and not numeric variables, so if you want to print a number (for example, the points on the scoreboard in your video game) you must do so:

```
points=points+1  
cad$= str$(points)  
|PRINTAT,0,100,10, @cad$
```

The |PRINTAT command is not affected by the limits for clipping set with |SETLIMITS. This is most logical since you will normally use PRINTAT to print scores on your markers, which will be outside the area bounded by |SETLIMITS.

Unlike the BASIC PRINT command, the |PRINTAT command is quite fast and can be used to update your game markers frequently.

PRINTAT uses a redefined alphabet, which may contain a reduced or different version of the "official" Amstrad characters. By default 8BP provides a small alphabet consisting of numbers, uppercase letters, white space and some symbols. You will not be able to use a character that is not in this set, such as lowercase letters. The characters of this alphabet are all the same size: 4 pixels wide x 5 pixels high, i.e. 2 bytes x 5 lines.

This string contains all the characters you can use with the 8BP serial alphabet. Note that there are no lower case letters and many symbols are missing, although you can create your own alphabet that contains them. The last symbol is the ":".

"0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ!:,.,"

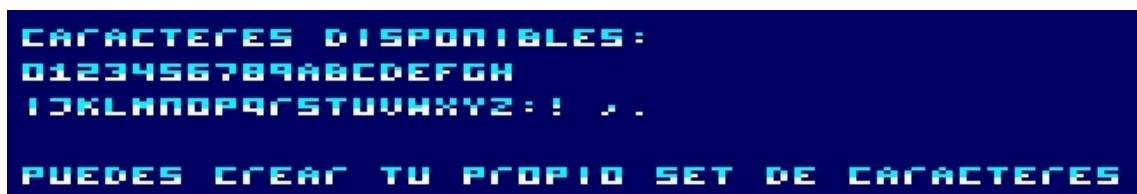


Fig. 96 default character set available in 8BP for use with PRINTAT

**IMPORTANT:** If you try to print with PRINTAT a character that does not exist in the created alphabet, the last character in the character list will be printed, which in the default alphabet is the dot "".

I have created the characters using inks 2 and 4, so that overprinting can be used, as the background colours are 0 and 1 and using overprinting ink 2 must be equal to 3 and 4 must be equal to 5 (see the chapter where I explain overprinting). To use overwrite simply set the transparency flag in the PRINTAT command to "1".

## 17.1 Create your own mini-character alphabet

The characters to be used with the PRINTAT command are defined in a file in the ASM directory and imported from the file "images.asm".

Let's look at a fragment of "images.asm" We will find these three lines:

if you are not going to use the |PRINTAT command, but just the Amstrad characters  
You can then comment on the following 3 lines

```
_BEGIN_ALPHABET  
read "alphabet_default.asm"  
_END_ALPHABET
```

The alphabet is a few pieces of data and the images of each character. All of this is assembled within the 8BP image memory area. The default alphabet is just over 400 bytes.

The file alphabet\_default.asm contains the alphabet that 8BP includes as standard. You can create your own alphabet file. This file contains 3 variables indicating the size of the characters, which you can draw in any size you want. By default I have created an alphabet of 2 bytes wide by 5 lines high, but you can decide to use another size, even create giant characters. If you change the width or height, you must also change the ALPHA\_SIZE variable in your alphabet.asm file accordingly.

**ALPHA\_width db 2; width alphabet.all letters measure the same**  
**ALPHA\_height db 5; alphabet height.all letters are the same**  
**ALPHA\_span db 2\*5**

Next, we find the string indicating the valid characters to use with the PRINTAT command. These characters are valid because they have an associated drawing. After this string, the drawings of all these characters are found one by one, in the same order in which they appear in the text string ALPHA\_LIST

```
ALPHA_LIST
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ! ,." ; characters created
db 0 ;zero byte indicates end of Alpha_list string
```

The pictures of each character in the ALPHA\_LIST string are shown below. I have created them with the SPEDIT tool. The first of them must be the first character of the ALPHA\_LIST string, i.e. "0".

The bytes corresponding to this character are shown here:

<pre><b>; character 0</b> <b>db 12 , 8</b> <b>db 8 , 8</b> <b>db 8 , 8</b> <b>db 32 , 32</b> <b>db 48 , 32</b></pre>	
--	--

Then we will find one by one the rest of the letters, numbers and symbols defined. With some patience you can create your own set of mini-characters, which will give more personality to your videogame. You only need to create the ones you are going to use, and the fewer there are, the less memory you will use in the image area.

Remember that if you try to print a character that you have not created, the last of the characters defined in the ALPHA\_LIST will be printed.

## 17.2 Default alphabet for MODE 1

The default alphabet of 8BP is created in MODE 0 and if you try to use it in MODE 1 it will not work properly, because they are drawings made in mode 0. By default 8BP also comes with an alphabet that works in MODE 1. You just have to alter one line of images.asm. This alphabet is inspired by a font called "5th agent".

```
_BEGIN_ALPHABET
read "alphabet_default_mode1.asm"
_END_ALPHABET
```

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ : ! .
```



## 18 Pseudo 3D

In the 1980s "Pole position" style car games were popular. They gave a 3D feeling, but they didn't do the 3D calculations, just for the road plan, and sometimes not even that, as they were approximations that gave a good feeling of speed.

In the arcade machines, specific chips were used to perform "sprite scaling", making the sprites increase in size smoothly, and the calculation of the road by means of a specific chip (such as the "sega Road chip") which was exclusively dedicated to painting the road with its stripes. The road map was then added to the sprite map and the final image was composed. These chips were used in arcade machines such as "Pole Position" and "Space Harrier".



*Fig. 97 Pole position and Out Run (arcade machines)*

The common feature of both software (used on 8bit computers) and hardware (used on arcade machines) techniques is that the curves are an illusion: the road is deformed while mountains on the horizon are moved to give the sensation of a turn, but there is no turn at all. The results were often very convincing, but the curves were really an illusion.

The techniques used on 8-bit computers were very varied. Anything was acceptable as long as the player was made to believe that he was on a racetrack. In many cases, ink rotation was used to give a sense of speed. Many games suffered from a low frame rate, below 5 fps. Among the best games for Amstrad were "**3D grand Prix**" (using software sprite scaling combined with ink animation), "**Buggy boy**" based on a very advanced programming technique of raster effect and some others like "**Crazy cars**" or "**Chase HQ**".

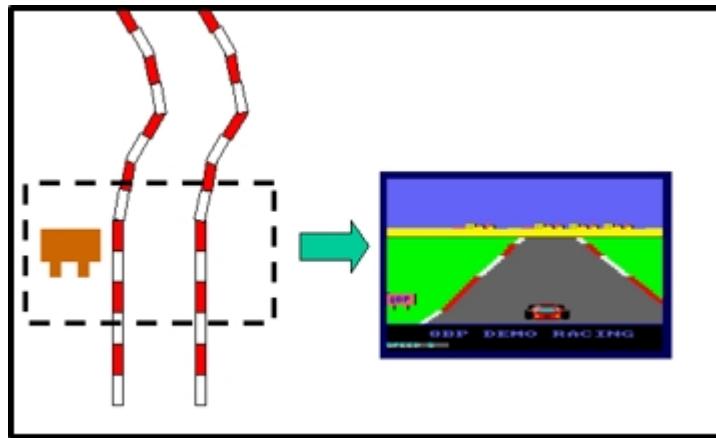
From version V32 of 8BP you have Pseudo-3D capability at your disposal, used in the demo game "**3D Racing one**". It is very easy to use and with it you can build your own racing games, tank games, ship games, etc.

To use Pseudo-3D in 8BP you must use "**assembly option**" 3, which leaves 24 KB free for your BASIC program.

The capabilities provided by 8BP to make pseudo-3D available are:

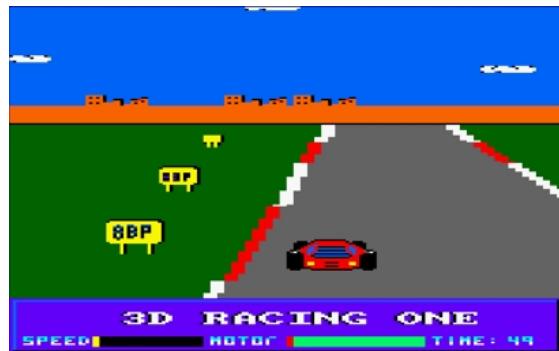
- 1) **3D projection:** this consists of projecting a 2D map of the world in 3D, so that, even if we go through it in 2D, it gives us the sensation of seeing it in 3D. This is done by invoking the |3D command to configure the PRINTSPALL commands.

and PRINTSP so that they project in 3D before painting on screen. There will be no possibility to rotate in the plane, we will always "look" forward, but the combined effect of a curve together with houses or mountains on the horizon moving, will simulate that we are taking a curve.



*Fig. 98 3D projection*

- 2) **Zoom:** this consists of being able to use different versions of the same image of an object to give a zoom effect as we approach it. This is simply done in the image file, defining 3 versions of the same image and grouping them in a structure. The image shows the typical example of the poster zooming in. The commands |PRINTSPALL and |PRINTSP will choose the most appropriate version of the image according to the distance at which it is located, without the need to do anything more than define the image as a "Zoom" type image.



*Fig. 99 Zoom images*

- 3) **Segments:** this consists of being able to have "segment" type sprites, defined by a single horizontal scanline (which means they take up very little space), to which we can associate a length and an inclination. With them we can build roads, rivers, etc. and roadsides. This is simply done in the image file by defining a type of image that has not only width and height but also slope. We will use these segments in the construction of the world map.



*Fig. 100 segments with different inclinations*

The segments used in the "3D Racing one" game are red or white and although they are long, they are only defined by a scanline. For example, the left white segment consists of a few green bytes for the grass, a couple of white and a few grey bytes for the road. At the time of printing, the segment thus defined is replicated to the desired length, and printed in perspective, so that even if it is a straight segment, it will appear crooked.

Finally, it is worth mentioning that in many occasions it will be necessary to dynamically update the map (command **|UMAP**). Thanks to this command, the map can be very large (which is necessary if we create a circuit with very many segments). This command is described in the scroll chapter.

In the following, we will take an in-depth look at these 3 features and how to use them in your programmes.

## 18.1 3D projection

To project we have the command |3D Use

**|3D, 1, <sprite\_fin>, <offsety> : REM this activates 3D projection.**  
**|3D, 0 REM disables 3D projection**

This command activates the 3D projection in the **|PRINTSP** command and in **|PRINTSPALL**. This means that before printing to the screen, the "projected" coordinates will be calculated and then printed to the screen. The coordinates of the sprites are not affected, i.e. the coordinates will remain the same, but in the 2D world. On screen we now have a 3D view and the coordinates where they will be printed will be the result of running the projection function on their 2D coordinates.

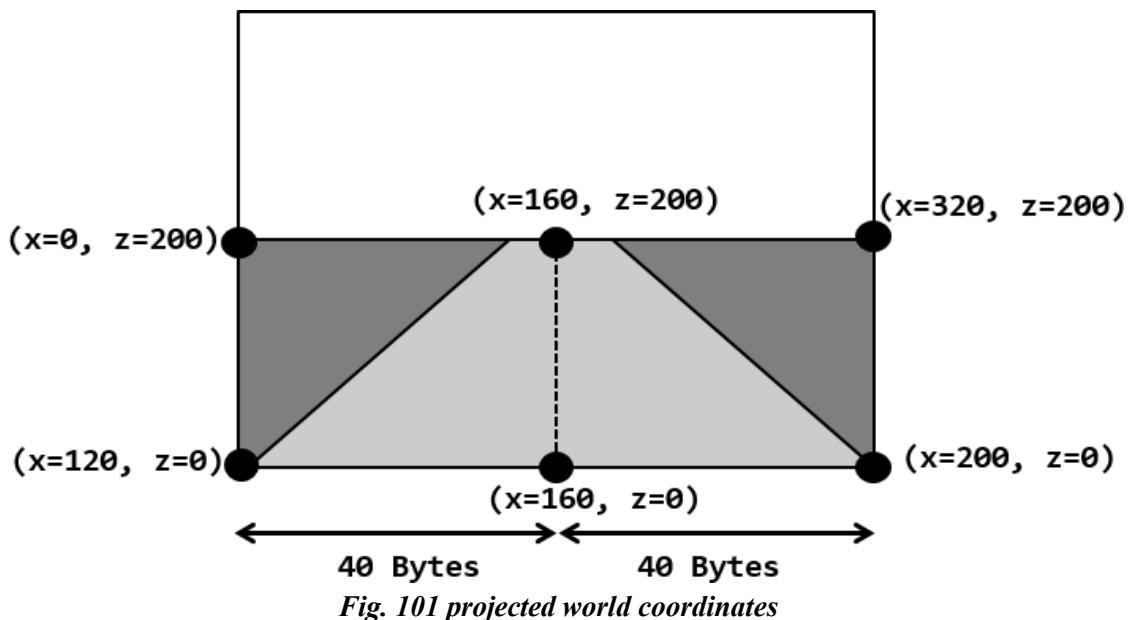
The affected sprites are all from Sprite 0 to **<sprite\_fin>**. The rest of the sprites will not be projected, they will simply be printed on screen in their 2D coordinates.

This command **does not affect collision mechanisms**, i.e. if we use COLSPALL and detect a collision between projected sprites, the collision will be occurring in the 2D plane. It may happen that in some cases two projected sprites partially overlap in the

screen, but that doesn't mean they have collided, one may be slightly ahead of the other and they may overlap when projected, but that is not a real collision. Collisions are detected in the 2D plane.

As for the last parameter <offsety> it is to project higher or lower, so we can place the game markers where we want. When projecting the screen, which is 200 pixels high, it becomes 100 pixels high, so we can choose how high we place the projection. If a Sprite is not affected by the projection because it is higher than <Sprite\_fin>, then it is not affected by <offsety> either. This is the case, for example, with the clouds and houses on the horizon in the game "3D Racing one".

To understand the screen coordinates on which your projected sprites finally appear, I recommend that you read the following very simple mathematics section to fully understand. The following figure represents what are the coordinates of the world map that are projected on certain representative points of the screen when MAP2SP is invoked with ( $yo=0$ ,  $xo=0$ ). The Z coordinate represents the distance at which the objects are located, also called "depth".



If instead of ( $xo=0$ ,  $yo=0$ ) we use another coordinate for MAP2SP, the 2D world coordinates corresponding to the points referenced in the image will be shifted by ( $x$ ,  $z$ ) as indicated by ( $xo$ ,  $yo$ ).

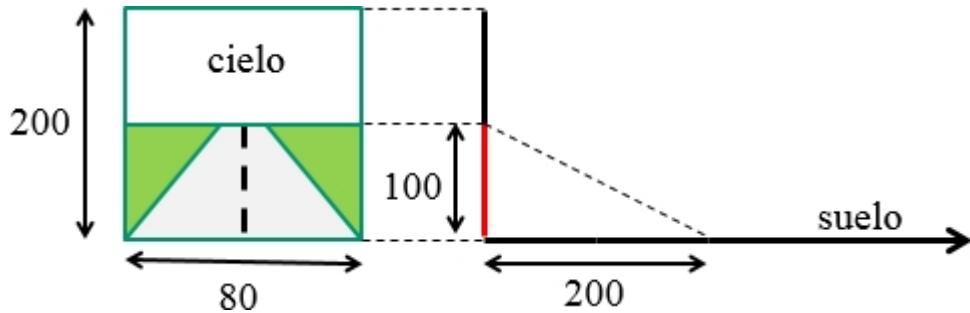
### 18.1.1 Mathematics of pseudo 3D projection

The pseudo 3D projection consists of projecting onto the screen the ground plane, which is where our 2D map of the world is.

#### Calculation of the Y-coordinate

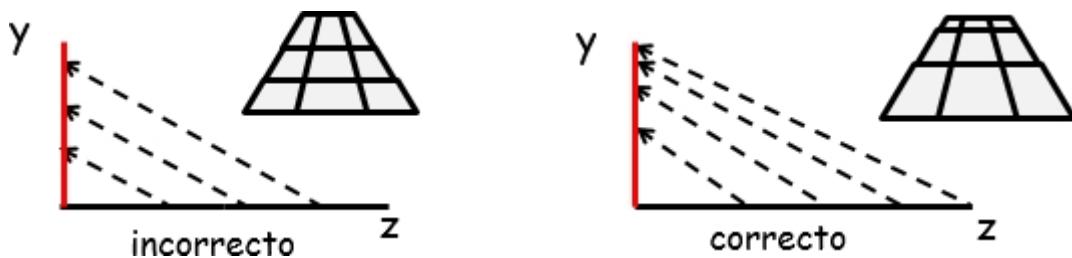
Our floor may be infinite, but we will project only 200 pixels long. This length is called "depth" or "Z" coordinate. These 200 pixels of depth, when projected, are compressed into 100 lines high (projected Y-coordinate). The pixels

The farthest projected objects are the horizon line. Note that we will not see very distant objects on the horizon, only those at a depth of 200 at the most.



*Fig. 102 projection of the floor on the screen*

As objects get further away they get smaller and smaller. It is not a linear relationship between the screen and the ground, i.e., to know how high to print a pixel that is at a certain distance, it is incorrect to think that as the depth covered is 200 and it is projected on 100 lines high, it is enough to divide by two. In a linear function (such as  $y = f(z) = A \cdot z + B$ ) the derivative ( $A$ ) is constant, but in the projection, what is constant is the "second derivative" of the function  $f(z)$ . We will now look at this in detail.



*Fig. 103 The correct projection is not linear*

Suppose we start with " $Z=0$ " and we want to know how much " $Y$ " is worth. It is very easy, at the ground line ( $z=0$ ) the  $y$ -coordinate is also zero.

If we increment  $z$  with a small variation " $dz$ ", the " $y$ " will be worth the previous " $y$ " (zero) plus an increment  $dy$ , which will initially be equal to the increment  $dz$  as the increment has been small and we are close to the horizon.

**dy (initial) = dz**

Now, if we again move away from  $dz$ , the increment  $dy$  must decrease, and if we again move away from  $dz$ , the  $dy$  to be added will be smaller and smaller. That is to say:

**Each time we move away from  $dz$ , we add an increment to  $y$  that each time we move away from  $dz$ , we add an increment to  $y$  that each time we move away from  $dz$ , we add an increment to  $y$ .**  
**time is less  $z=z+dz$**   
 **$dy = dy + ddy$  , where  $ddy$  negative  $y= y + dy$**

The increase in dy is constant. We have called this increment ddy. Well, dy is the "derivative" of y, while "ddy" is what is called the "second derivative". Here the derivative is not constant, but the second derivative is.

The constant value we assign to "ddy" will produce more or less exaggerated projections. In 8BP the ddy value is negative, about -0.005, making the dy at the ground line almost 1, while at the skyline the accumulation of 200 ddy makes the dy value end up at zero.

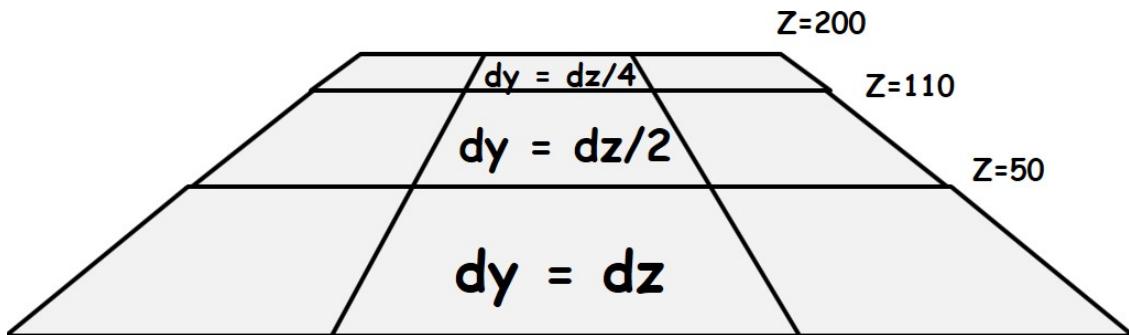
Despite the simplicity of these calculations, doing them in real time is costly for our beloved Amstrad CPC, so 8BP actually makes an approximation to avoid calculations and translate "z" to "y" in a much simpler way and with a very similar result.

If  $0 \leq z < 50$ , then  $dy = dz$ , therefore  $y = z$

If  $50 \leq z < 110$ , then  $dy = dz/2$ , so  $y = 50 + (z-50)/2$

If  $110 \leq z \leq 200$ , then  $dy = dz/4$ , so  $y = 50 + (110-50)/2 + (z-110)/2$

That is, we have divided the screen into three strips and each strip is treated as a "linear" area (since "dy" is constant) but with a different value of "dy" with respect to the other strips. This approximation gives visually very similar results to the mathematically correct ones.



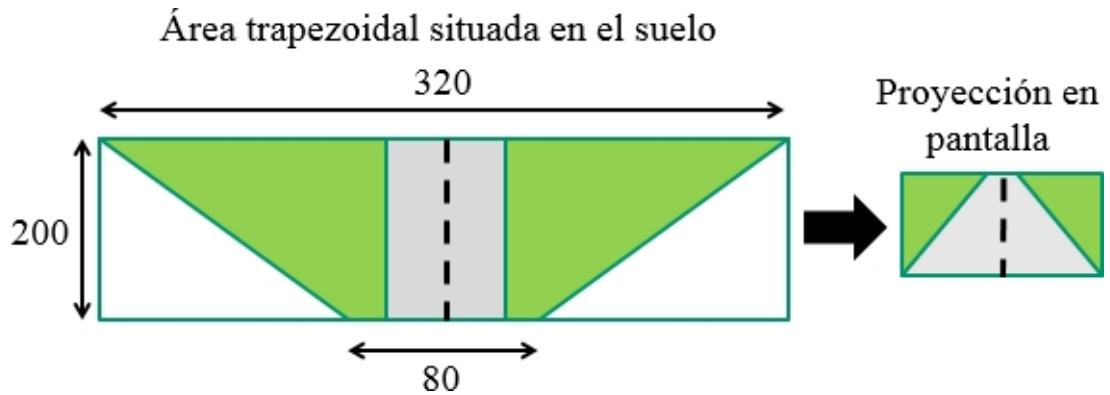
*Fig. 104 Approach of 8BP*

The equations used in 8BP also take into account that the screen coordinates are actually "inverted", i.e. the zero coordinate is the top coordinate and the 200 coordinate is the bottom coordinate, but this is just a very simple final adjustment.

#### **Let us now turn to the calculation of the "X" coordinate:**

There is a fundamental difference between the horizon line and the ground line. The ground line measures 80 bytes, but the skyline represents more width, because the road is straight and what measures 80 on the ground, measures a fraction in the skyline, specifically in 8BP it measures 4 times less. The road narrows at the horizon, because the horizon represents much more. In the projection applied by 8BP it represents 320 bytes.

And if the horizon measures 320 and the ground 80, it is because the total real area that we are able to see on the screen once the projection has been made is a trapezoidal area. It is NOT that the ground is a trapezoid, but that the area of ground that we are able to see on the screen is trapezoid-shaped.



*Fig. 105 Displayed trapezoidal area*

Intuitively and without showing any equations, it is already clear what we want to do and how the projection works. Now let's look at the equations.

In essence the mathematics of the "X" coordinate is the same as that of the "Y" coordinate. However, they are not done in the same way, because once we have the "Y" coordinate calculated, we can make a linear equation  $x=f(y)$  because due to the non-linearity of "Y" with respect to "Z", it is like creating a non-linear relationship between "X" and "Z". Earlier we said that the horizon is 320 bytes long and the soil is 80 bytes long. This means that the ground is 4 times smaller than the horizon. In the far distance we must divide by 4 (dividing is expensive so we prefer to multiply by a factor 0.25) while in the near distance we must multiply by a factor = 1.

What we have to do is to centre the X coordinate of the object to be projected with respect to the centre of the screen. We will then multiply by a factor that will depend on the projected "Y" coordinate. This will establish a non-linear relationship between "X" and "Z".

We must construct an equation that returns a result 4 times smaller on the horizon, for example:

```

Factor = ((100-y)+ 32 ) / 2
x= (x - centre) * Factor + centre
if z=200 then y= 100, and then factor =16
if z=0 then y= 0, and then factor =64 ( 4 times more)

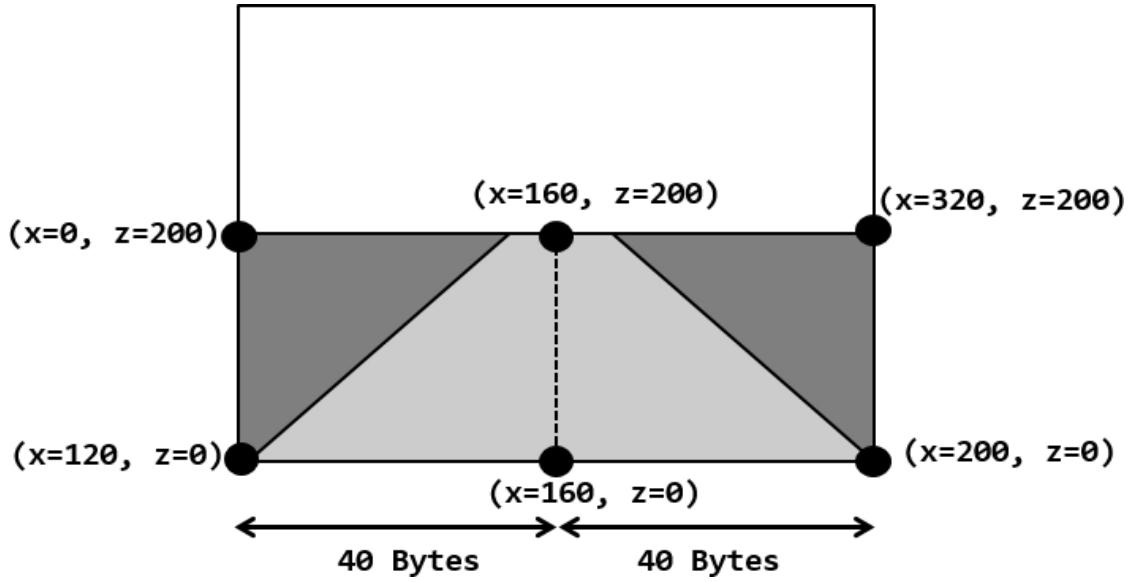
```

The chosen equation is interesting because the division by 2 is something that the Amstrad can do by rotating a bit in binary. That is, it can do it in a few clock cycles.

As shown in the equation, to project the "X" coordinate, it is sufficient to centre it and then multiply it by the factor obtained. The number thus obtained is very large, because in the case of the horizon line we will be multiplied by 16 and in the soil line by 64, i.e. we have multiplied the x-coordinate by a factor whose value is between 16 and 64. Between the horizon and the ground we will have all the 48 possible values for the factor, so, although the factor is an integer, it will evolve smoothly.

Then we divide by 64, which is just rotating in binary 6 times, and that's it. The latter will be equivalent to having multiplied by 0.25 the horizon, by 1 the ground and by the decimal value corresponding to any intermediate height between the ground and the horizon, but we have done it with integers, which the Amstrad can handle fast. This technique is called "fixed point arithmetic".

Taking all this into account, and supposing we ask MAP2SP to generate the sprites of the world map from the coordinate ( $yo=0$ ,  $xo=0$ ) what we will be seeing on the screen will have the following coordinates of the world. I'm sure it's easy to understand the figure now.



*Fig. 106 Projected world coordinates*

As you can deduce, the point ( $x=0$ ,  $y=0$ ) of the world map is projected off the screen, it is not displayed. If instead of ( $xo=0$ ,  $yo=0$ ) we use another coordinate for MAP2SP, the 2D world coordinates corresponding to the points referenced in the image will be shifted by ( $x$ ,  $z$ ) as indicated by ( $xo$ ,  $yo$ ).

### 18.1.2 Curves

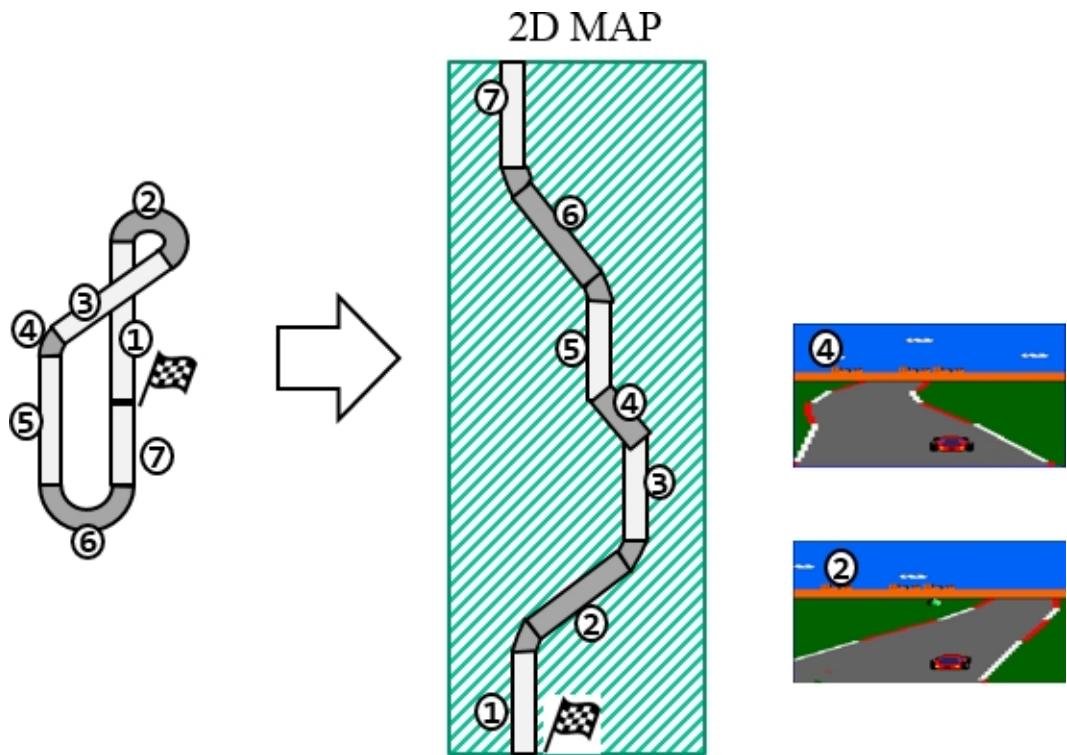
As I mentioned at the beginning of this chapter, the 3D projection used by 8BP does not allow rotations of the plane, so to simulate a curve we will have to do a little trick. This involves twisting the road to the right or left, while moving sprites such as houses or mountains on the horizon. These sprites will not be projected and to avoid this we will use identifiers above `<Sprite_fin>`. Remember that to activate 3D projection we will do:

**|3D, 1, <Sprite\_fin>, <offsety>, <offsety>, <offsety>, <offsety>, <offsety>.**

Consequently, an apparent circuit with curves will be represented on our 2D map as a road with slopes to the right and left, simply.

With this strategy, we can create the sensation of a racing driver driving around a circuit with real curves, and give the impression that his car is turning in the curves by means of houses or mountains on the horizon that move in the opposite direction to the X-coordinate. If you are a good artist and gradually twist different segments to a greater or lesser degree, you give the impression of very realistic curves.

This strategy is computationally very efficient and sufficiently realistic. If we wanted to rotate the ground plane for real, we would have to apply matrix computation to rotate, with many very expensive operations and, in addition, the sprite textures on the ground would have to rotate as well, so the computational cost would be enormous.



*Fig. 107 imaginary circuit and 2D world map*

Today's computers are so powerful that the strategies presented here are meaningless, but they are strategies superior in elegance and ingenuity to today's "brute force". Remember that "limitations are not a problem, but a source of inspiration".

You will need to use the **|UMAP** command to traverse large circuit maps, but remember to invoke **|UMAP** not every cycle but only from time to time.

## 18.2 Zoom images

To define a "**zoom image**", we simply create the different versions of the image (3 versions). Then, in the image file "images\_mygame.asm" we will look for a tag called "**\_3D\_ZOOM\_IMAGES**".

We will find this:

```

_BEGIN_3D_ZOOM_IMAGES
;=====
;; limits applicable to all zoomed images
The horizon for these limits is considered to be 0 and increasing
downwards to 200
_ZOOM_LIMIT_A
db 120; between 200 (ground) and limitA is set to image 3
_ZOOM_LIMIT_B
db 50
between this boundary and boundary A, image 2 is set.
closer to the horizon than limit B is set image 1
;=====
```

```

CARTEL_ZOOM
db 1; width symbolic db
1; height symbolic
dw POSTER1, POSTER2, POSTER3

_END_3D_ZOOM_IMAGES

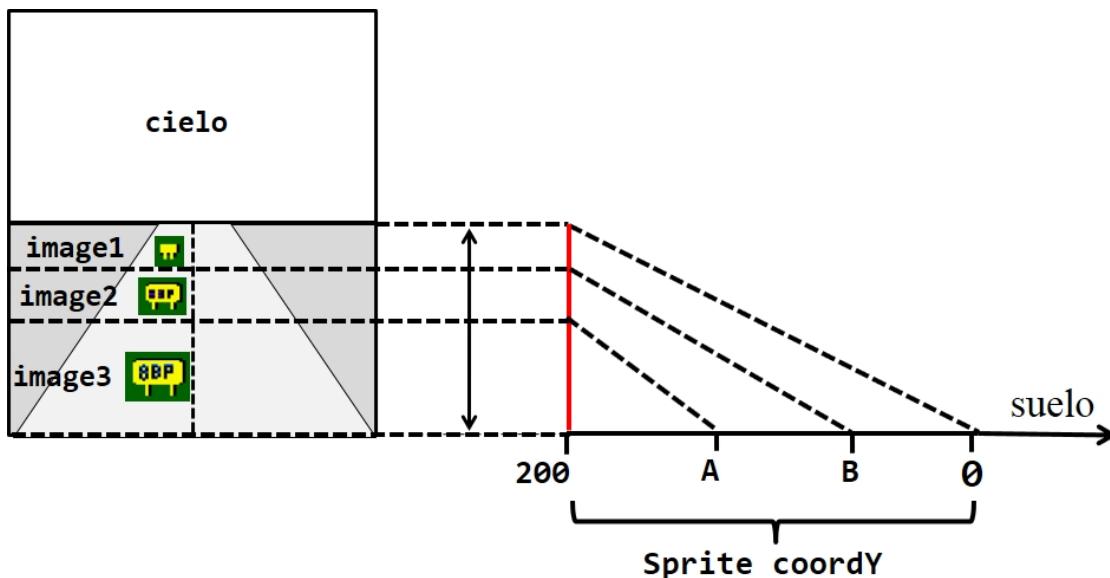
```

All ZOOM images must be defined after the "**\_BEGIN\_3D\_ZOOM\_IMAGES**" tag. These are images that have a symbolic width and height, because in reality a Sprite to which one of these images is assigned, will use the width and height of the version of the image that is automatically chosen according to its Y-coordinate. That is, "CARTEL1" is a normal image, previously defined, with its width, height and bytes containing the drawing. And the same can be said for "CARTEL2" and "CARTEL3". If we associate the image "CARTEL\_ZOOM" to a Sprite (this can be done by associating an identifier to this image at the beginning of the image file) what will happen is that, depending on the position of the Sprite on the screen, one or other version of the image will be printed.

For the automatic selection of the image, y-coordinate thresholds are set. You can change these thresholds, but the default ones work fine and they are:

- between horizon =0 and 50, the first image is chosen (in the example, "CARTEL1").
- between 50 and 120, the second image is chosen (in the example, "CARTEL2").
- between 120 and 200, the third image is chosen (in the example, "CARTEL3").

The choice of these limits to delimit the screen borders is configurable and you can set as many as you want. Note that the choice is made based on the Y-coordinate of the unprojected Sprite. Once projected, its Y coordinate varies a lot, but it is not the projected "Y" that is used to delimit the 3 stripes, but the "Y" of the Sprite in 2D. When the Sprite moves from one stripe to another, its image will change automatically. If you prefer to have an image appear first, you can modify the thresholds.



*Fig. 108 Usage ranges of the 3 versions of the ZOOM image*

For example:

```
REM let's assume that CARTEL_ZOOM has id=16 in the image file
|SETUPSP,20,9,16 : REM associates CARTEL_ZOOM to Sprite 20
|LOCATESP, 20,100, 160: REM Sprite at at centre of the
    "trapezoid
(coordy=100)
|3D,1,31,0: REM all sprites will be projected
|PRINTSP,20: REM version 2 of the POSTER is printed.
```

### 18.3 Use of segments

Now that you know how to build a 2D map that "simulates" a circuit with curves, here is a powerful way to build the segments with different degrees of inclination that you need to build racing circuits or roads.

A segment is a single line high image, which by repetition can reach any length we want. In addition to the length, it has another parameter, which is the total slant of the segment, in bytes. This slant can be negative (slant to the right) or positive (slant to the left).

To define this type of images you must create them in your game's image file "images\_mygame.asm", after the tag "\_BEGIN\_3D\_SEGMENTS".

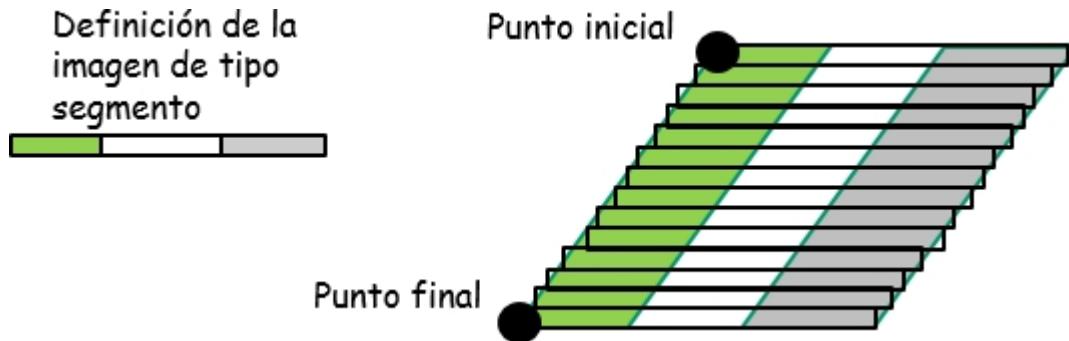
```
;=====
;----- _BEGIN_3D_SEGMENTS -----
;----- There could be a different definition of segments.
;----- the width is the width of the scanline
;----- the high is the 2D high of the segment
;----- then goes the dx, which can be positive (tilted left) or negative
;----- (tilted right).
db 0; this is for the first segment type image to be >
_3D_SEGMENTS

;----- SEGMENT_EDGE_LWI10 -----
SEGMENT_EDGE_LWI10
db 22; width
db 50; high
db 10; dx
db 192,192,192,192,192,192,192,192,192,192,192,192,192,192,192,240, 240
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
;-----
```

In the example we have defined a segment that has grass on the left (green bytes), then two white bytes and then grey bytes (road) on the right. Whether it is green or grey depends on the colours we have associated with the inks.

It is a segment with a slant to the left of 10 bytes. If we had put a zero in the slant (dx), it would be a straight segment, not skewed. It is 50 lines high, but when projected it is less, unless it is very close.

The points of the segment that are projected are only two. Once projected, the scanlines are painted one by one with a certain horizontal displacement so that the last line ends up starting at the end point. Note that, although the segment is painted in perspective, its width is constant. You do not paint the top part narrower (further away) and the bottom part wider (closer), but you paint each scanline exactly as it has been defined in the image file.



*Fig. 109 2 points are projected on a segment*

As you can see I have used a lot of grass and road bytes. This is so that the segment "erases itself" as it moves forward, although if the car is going too fast, traces may remain. Once you get your game running you will check if the speed is too fast and traces are left.



*Fig. 110 Relationship of the degree of inclination of a segment and the maximum speed*

As can be seen in the figure, the maximum feed rate for a self-erasing segment can be higher if the degree of skew is moderate. If it is very crooked, the speed cannot be as great, or traces will remain. Once the segment is projected, it will be shorter because of the perspective effect, and depending on where it is, it can be even more or less crooked. It is advisable to make them wide so that they erase themselves more safely.

In the game "**3D Racing one**" there is a stage called "superfast" in which, using less bent segments, I increased the speed of the car without the segments leaving any traces. A simple and very effective "trick". If the game is slow (e.g. a tank game, which is supposed to be slow) then the segments can be very crooked as they will not leave a trace due to the speed effect.

In short, to increase speed you have two options:

- Use less twisted segments
- Use wider segments (with more margin to erase themselves).

## 19 Music

The tools I am going to talk about in this section are not programmed by me, but they are integrated in 8BP and they are really good.

### 19.1 Editing music with WyZ tracker

This tool is a music sequencer for the AY3-8912 sound chip. The music it generates can be exported and results in two files

- An instrument file ".mus.asm".
- A file of musical notes ".mus".

You can compose songs with this tool and the only limitation is that all the songs you integrate into your game must share the same instrument file.

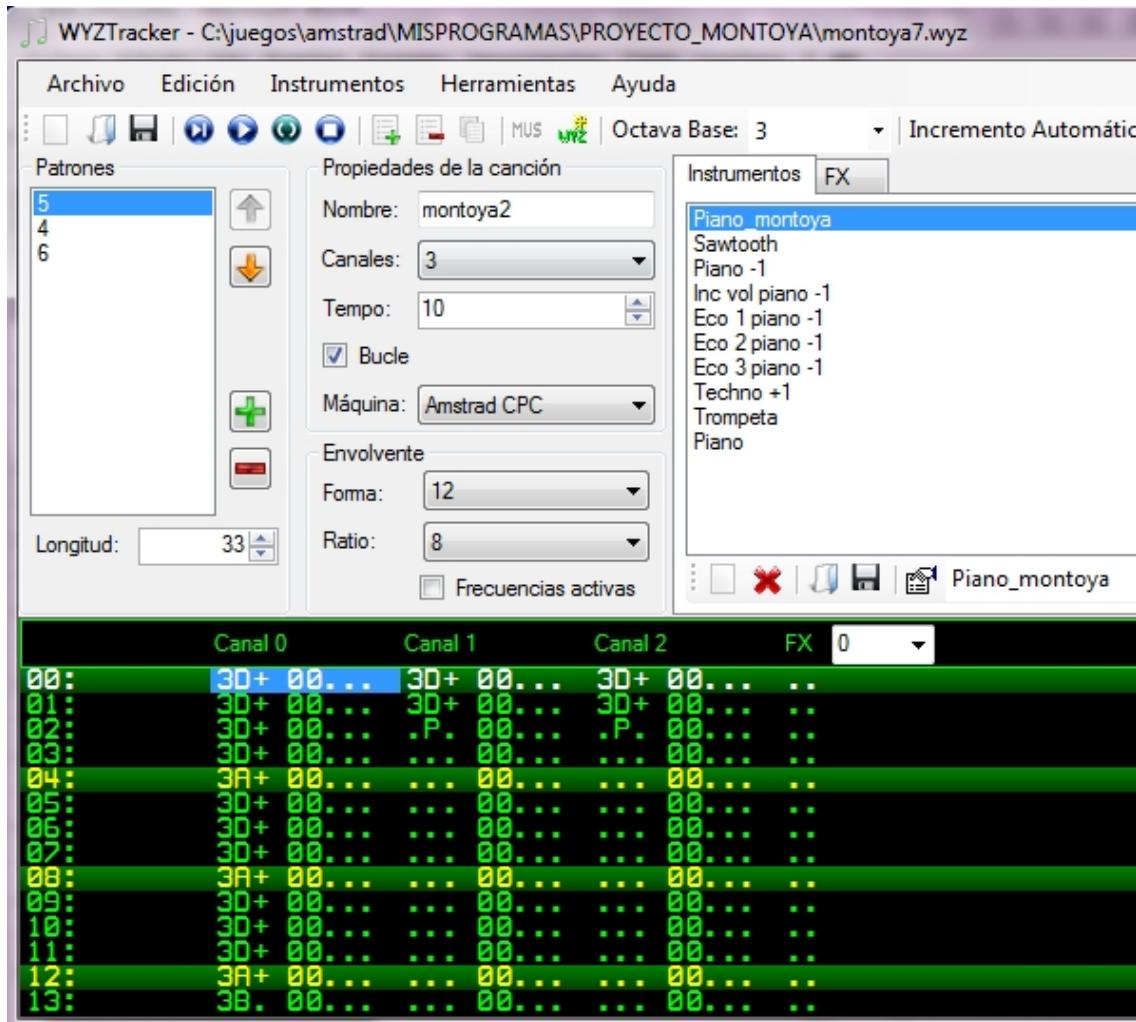


Fig. 111 WYZTracker

Since version V41 you can use the "FX" channel to incorporate sound effects. Until this version this pseudo channel was ignored in the 8BP player. Be careful when editing the music because, although the FX channel is always associated to a channel, inside the WYZ tracker it sounds as if it was independent and the result can be different from when you hear it in the WYZ tracker.

If the FX is associated to channel 0 and you put an FX at a time when channel 0 is not playing, nothing will be heard in the amstrad even if your effect is heard in the WYZtracker.

This music sequencer is complemented by the WYZplayer which is integrated in the 8BP library.

Since version V26 of 8BP, music can be composed with WYZtracker version 2.0.1.0 and it works really well. Until version V25 of 8BP the compatibility was with WYZtracker 0.5.07 and there were some small problems, but all that has disappeared with WYZtracker 2.0.1.0.

**IMPORTANT:** do not use octave 7 even if the tracker allows you to use it. This octave is not able to be played and synchronization failures may occur during music playback on the Amstrad.

An important recommendation when creating your songs with WyZtracker is to delete the instruments that you are not going to use. This way the instrument file (which ends in ".mus.asm") will take up less space, and since 8BP only reserves **1,500 bytes** for the music, every byte is important.

**IMPORTANT:** if you edit the music with a version of WYZtracker later than the 2.0.1.0 may not work properly and you may experience strange effects such as a channel not playing or notes being out of sync. If this happens to you, the solution is to create a file from scratch with WYZtracker 2.0.1.0 and manually copy note by note the music that doesn't work.

## 19.2 Assembling the songs

Once you have composed your song and you have the two files, you simply edit the make\_music.asm file and include your music files like this:

```
after assembly, save it with save "musica.bin",b,32200,1400

org 32200
;-----MUSICA-----
The limitation is that you can only include a single file of
; instruments for all songs
The limitation is solved by simply putting in all the
instruments in a single file.

instrument file. NOTE IT MUST BE ONLY ONE read
"instruments.mus.asm"

; music files SONG_0:
INCBIN      "micancion.mus" ;
SONG_0_END:

SONG_1:
```

```

INCBIN      "another_song.mus"
; SONG_1_END:

SONG_2:
INCBIN      "third_song.mus"      ;
SONG_2_END:
SONG_3:
SONG_4:
SONG_5:
SONG_6:
SONG_7:

```

Finally, you reassemble the 8BP library so that the music player (which is integrated in the library) knows the instrument parameters and the place where the songs have been assembled.

To do this you simply assemble the file make\_all.asm, which looks like this

```

; Makefile for video games using 8bit power
if you alter only one part, you only have to assemble the corresponding
make
for example you can assemble the make_graphics if you change drawings
;-----CODIGO -----
;includes the 8bp library and the music playerWYZ
read "make_codigo.asm".

;-----MUSICA-----
; includes the songs. read
"make_musica.asm"

; ----- GRAPHICS -----
This part includes images and animation sequences.
and the sprite table initialised with those images and sequences read
"make_graphics.asm".

```

And with this you have everything assembled. Now you must generate your 8BP library like this:

**SAVE "8BP.LIB", b, 24000, 8200**

And the music:

**SAVE "music.bin", b, 32200, 1400**

### **19.3 What to do if you can't fit music in 1400 bytes**

It is possible that 1400 bytes are not enough for your songs. In case a song doesn't fit (and you will know this by checking where the "\_END\_MUSIC" tag is assembled) you can specify a different assembly address for that song and the following songs. In the case of the "Nibiru" videogame, you do this with the third song, assembling it at a lower address than the 8BP library (e.g. 23000 would work). In case you do this, your BASIC game will have to start with a

MEMORY command specifying an address lower than this new limit, e.g. MEMORY 22999 would work.

From 8BP version 34 onwards, the library is assembled from address 24000, so if you want to use extra space for music, it must occupy addresses lower than 24000. For example, from 23500 to 23999.

This is the

```
after assembly, save it with save "musica.bin",b,32200,1400
org 32200
;-----MUSICA-----
; has the limitation of only being able to include only one file of ;
instruments for
The limitation is solved by simply inserting all the songs.
instruments in a single file.

instrument file. NOTE IT MUST BE ONLY ONE read
    "../MUSIC/nibiru5.mus.asm" ;

; music files SONG_0:
INCBIN      "../MUSIC/attack5.mus" ;
SONG_0_END:

SONG_1:
INCBIN      "../MUSIC/nibiru5.mus" ;
SONG_1_END:

org 23500 ; I'm using this line because I can't fit the third song !!!

SONG_2:
INCBIN      "../MUSIC/gorgo3.mus" ;

SONG_3:
SONG_4:
SONG_5:
SONG_6:
SONG_7:
_END_MUSIC
```

The same type of solution is applicable in the case where you can't fit all your graphics in the area reserved by 8BP, although as you have 8540 bytes for graphics you are less likely to have this problem.

## 20 C programming with 8BP

At the beginning of this manual I recommended you not to use BASIC compilers like Fabacom or CPC BASIC 3 because of the memory limitations they impose (you lose about 20KB). If what you want is to increase the speed of your games, from 8BP v40 onwards you have at your disposal a wrapper of the 8BP library to be used from C, and a small set of BASIC commands invokable from C (which I call "minibasic") so you can translate your BASIC program almost immediately and get the fast result you are looking for.

With this new functionality you have 3 options:

- 1) **Make your program 100% in BASIC** (i.e. do not use the functionality). This option simplifies a lot the programming task but you have less speed.
- 2) **Make your program 100% in C**. This option is complex because programming, compiling, searching and correcting errors is a much slower task than programming in BASIC.
- 3) **Make your program in BASIC and at the end translate only the game cycle to C**. This option is as easy as the first one except for the last phase of C translation.

I'm going to explain the third option, which is very interesting because it allows you to program in BASIC with the convenience it has (easy to program, to detect and correct errors, etc.). If you want to program your game entirely in C, this explanation serves you exactly the same, so don't be afraid and continue reading. A commercial example of option 3 is the famous and mythical game "galactic plague", an Indescomp production created by the excellent programmer Paco Suárez in 1984. We owe many hours of entertainment to the great Paco Suarez.

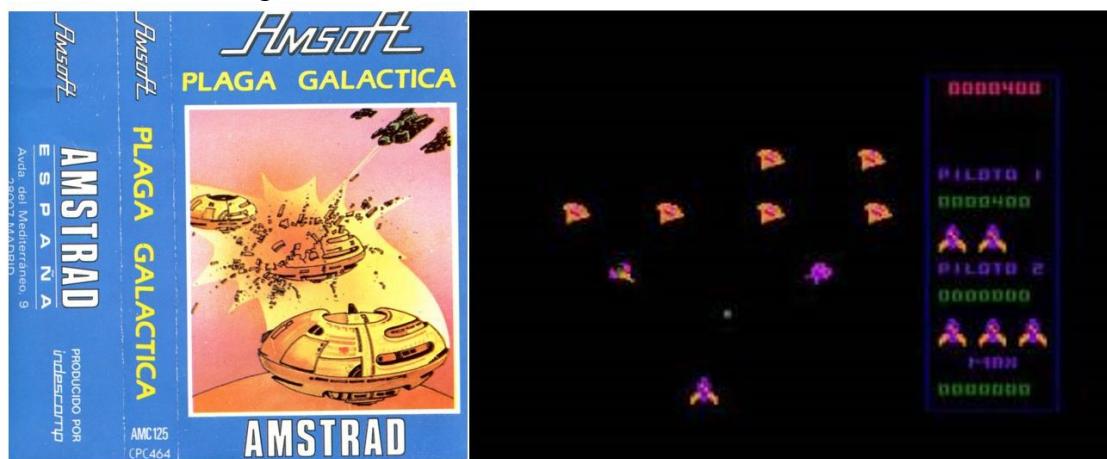


Fig. 112 The "galactic plague".

To be able to program in C we need some tools, but don't worry you don't have to learn how to use them because 8BP does it for you, thanks to a .bat that takes care of everything as you will see soon. The tools are:

- **compila.bat**: is the .bat file that calls the different tools in cascade and from a .c file you get a .dsk file with a binary inside. You will find this tool in the subdirectory "C" of 8BP.
- **SDCC** (Small Device C compiler): you must download and install it. You will find it at <http://sdcc.sourceforge.net/>. This is possibly the best C compiler for Z80 (although SDCC also supports other microprocessors). There is another one

I chose the Z88dk but I preferred to choose the SDCC because some tests reveal that the resulting compiled program with SDCC is faster than the z88dk equivalent.

- **Hex2bin.exe:** we will use it (from the .bat) to convert the file we generated in hexadecimal with SDCC to binary. No need to download it, it is small and you will find it in the subdirectory "C" of 8BP.
- **ManageDsk.exe:** we will use it (from the .bat) to put our compiled binary into a .dsk file. You don't need to download it. It is small and you will find it in the subdirectory "C" of 8BP.

Let's go through the necessary steps with an example and then I will detail how to invoke each of the 8BP functions from C, as well as the new "minibasic" commands that 8BPV40 provides you with to program in C as if you were in BASIC.

## 20.1 First step: program your BASIC game

We are going to use the example program that comes with the 8BP library. It is a very simple game in which you play a soldier who has to dodge balls falling from the sky in different directions. Each time a ball hits you, you lose points, which grow over time.



*Fig. 113 The example set*

The list of the game is as follows, in which I have highlighted in red the part that corresponds to the "game cycle".

```

10 MEMORY 19999
11 LOAD "loop.bin",20000: REM load compiled game loop
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
30 ON BREAK GOSUB 320
40 CALL &BC02:'restore default palette just in case'.
50 INK 0,0: 'black background
60 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:|3D,0:'reset sprites
70 |SETLIMITS,0,80,0,124: ' set the limits of the game screen
80 PLOT 0,74*2:DRAW 640,74*2
90 x=40:y=100:' coordinates of the character
100 PRINT "SCORE:      FPS:"

110 |SETUPSP,31,0,1+32:' character status
120 |SETUPSP,31,7,1'animation sequence assigned at startup
130 |LOCATESP,31,y,x:'place the sprite (without printing it yet)
140 |MUSIC,0,0,0,0,5: points=0

150 cor=32:cod=32:|COLSPALL,@cor,@cod:' set collision command
151 LOCATE 1,20:INPUT "basic(1) or C(2)", a: IF a=1 THEN 160 ELSE CALL &56b0

```

```

152 GOTO 320
160 |PRINTSPALL,0,0,0,0,0: 'configure print command
161 POKE &B8B4,0: POKE &B8B5,0: POKE &B8B6,0: POKE &B8B7,0:'reset timer
cpc6128. it is necessary because TIME can return a very large number and can
give overflow with DEFDINT
162 t1=TIME
170 --- game cycle. This is the part that has been translated into ---
C
180 c=c+1
190 ' reads the keyboard and positions the character
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:dots=dots-1:LOCATE 7,1:PRINT           points:GOTO
250
221 REM to calculate the FPS we take into account that TIME gives me in units 1/300
seconds and I am going to measure every 20 cycles. Therefore fps= 20 cycles x 300
/ dt, where dt= t2-t1
230 IF c MOD 20=0 THEN dots=dots+10 :LOCATE 7,1:PRINT dots:
t2=t1:t1=TIME:fps=6000/(t1-t2):LOCATE 17,1:PRINT fps
240 IF c MOD 5=0 THEN |SETUPSP,i,9,9,19:|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 game cycle ---
252 |POKE,42038,points
310 end of the game
320 |MUSIC:INK 0,0:PEN 1:BORDER 0:|PEEK,42038,@points
330 LOCATE 3,10:PRINT "FINAL SCORE:";dots

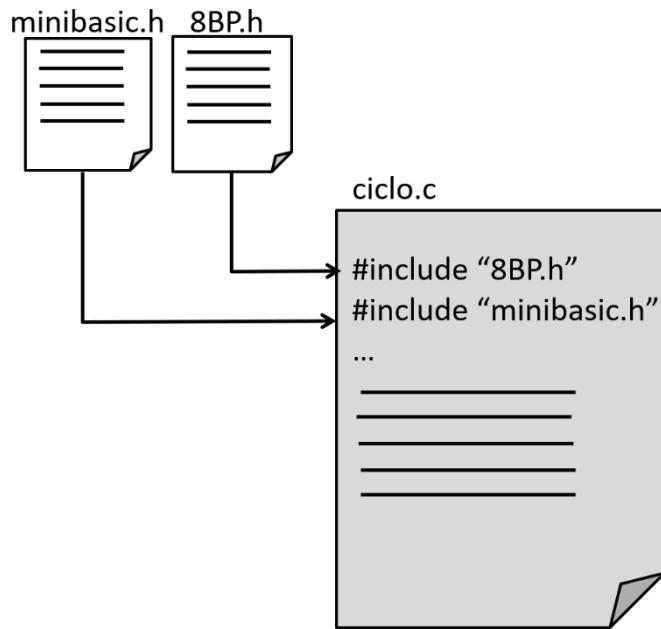
```

## 20.2 Step 2: translate your BASIC game cycle into C

To translate the game cycle into C, we need to write a C program, which we will call `cycle.c`.

Go to the subfolder "C" of 8BP. There you will find everything you need and this same example, with the file `ciclo.c`.

The first thing you need to know when programming the game cycle in C, is that you have two small libraries to include: the 8BP wrapper (`8BP.h`) and the minibasic (`minibasic.h`). The `8BP.h` is in the subdirectory "`8BP_wrapper`" and the `minibasic.h` is in the subdirectory "`mini_basic`".



*Fig. 114 files to be compiled*

This is the C listing, which as you can see has a direct correspondence with the piece of BASIC listing **corresponding to the game cycle**, practically a literal translation. You will see some labels like "label\_195" that act as line numbers to be able to jump with GOTO. It is practically the BASIC list translated instruction by instruction, without the need to rethink how to program it. In this simple case there is only one function (the main function) and it returns when you run out of time.

To take this step you have the "minibasic" to help you in the translation from BASIC to C, but if you are an expert in C and you know the AMSTRAD firmware or you have some other help library you can make the game cycle directly in C, without having previously programmed and validated it in BASIC. The way I propose you is easy and powerful, your program will run like a greyhound, but here you are free to do it as you want.

**IMPORTANT:** remember when programming in C that the notation for decimal, hexadecimal and binary numbers is:

```
mivariable = 165 ; //decimal notation
mivariable = 0xA5 ; //hexadecimal notation
mivariable = 0b10100101; //binary notation
```

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stdio.h>

#include "8BP.h"
#include "minibasic.h"
#include "8BP.h"
#include "minibasic.h"

//declare the variables all global in order to be able to
//access them from any function, like in BASIC
// although they are not initialised here
//-----
int c;
```

```

char dir;
int x;
int y;
int cod;
int cor;
int i;
int points; int
t1;
int t2;
int fps;
*****
MAIN
*****
int main()
{
    //initialise the variables c=0;
    dir=0;
    x=40;
    y=100;
    cod=32;
    cor=32;
    i=0;
    points=0;
    fps=0;
    t1=_basic_time();

    //configure commands
    _8BP_printspall_4(0,0,0,0);
    _8BP_colspall_2(&cor,&cod);

    //game cycle
    //-----
    label_CICLE:
    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(dots)); goto
    _label_250;

}
else _basic_border(0);
if (c %20 ==0) {
    points=points+10;
    _basic_locate(7,1);
    _basic_print(_basic_str(points));
    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_CICLE;

    _8BP_poke_2(42038,points);
    return 0;

```

}

As you can see the 8BP functions are invoked as:

**\_8BP\_<function>\_<N>( parameters)**

N being the number of parameters of the function as there are versions of each function with different number of parameters (as happens in the RSX versions of the commands).

And basic functions that are not available in C but that we have created in minibasic are invoked like this:

**\_basic\_<function>(parameters)**

As you can see, knowing the translation of each command it is very easy to translate a BASIC listing of your game cycle into a cycle.c listing.

A necessary thing to do is to communicate between LOCOMOTIVE BASIC and C. For example, there are data that you may want to pass to the C program, such as the number of lives you have left or the points you have accumulated. For that you can use the functions:

- From LOCOMOTIVE BASIC you have **PEEK**, **POKE**, **|PEEK** and **|POKE**.
- From C you have **\_basic.Peek()**, **\_basic.Poke()**, **\_8BP.Peek\_2()**, **\_8BP.Poke\_2()**

With these functions you can reserve a memory address to store the lives, points, phase, etc. and thus invoke the game cycle every time you are killed with all the game context in a few variables that can be read and modified by both BASIC and C. In the example you can see how this is done with the variable "points".

**IMPORTANT:** even if your whole program is in C, you must initialise 8BP with a **CALL &6b78**, because apart from installing the RSX commands, this call also initialises the tables of the internal library.

### 20.2.1 GOSUB and RETURN in C

GOSUBs should be translated into c functions because you can get to a GOSUB routine from anywhere in the program and you must be able to return. In BASIC you return with RETURN to the place from where you invoked GOSUB. In c the natural thing to do is to translate it to a function, to be able to return to the point of the program from where you called it

Let's look at an example:

BASIC	C
<b>10 a=5</b> <b>20 GOSUB 100</b> <b>30 PRINT "PEPE"</b> <b>40 end</b> <b>100 REM ROUTINE</b> <b>110 PRINT STR\$(a)</b> <b>120 RETURN</b>	#include <stdlib.h> #include <string.h> #include <stdio.h> #include <stdio.h>  <b>#include "8BP.h"</b> <b>#include "minibasic.h"</b>

	<pre>#include "8BP.h" #include "minibasic.h"</pre>
	<pre>void mifucion(int id); int a; int main() { a=5; mifunction(a); _basic_print(_"PEPE");  return 0; }  void mifucion(int a) { _basic_print(_basic_str(a)); _basic_print("\r"); }</pre>

### 20.2.2 BASIC to C communication with BASIC variables

If you are just starting to use C with 8BP you can move on to the next step. This is an "advanced" section to teach you how to communicate between BASIC and C with BASIC variables instead of PEEK/POKE, but it is not essential.

To communicate BASIC with C, instead of a memory address and PEEK/POKE, you can also use a variable that exists in BASIC. It's a bit more complex, but it can be done. Let's see how to do it with a simple variable, and then we'll see how to do it with array variables.

The first thing you need to know is how to find out where BASIC stores a variable. For this we have the "@" operator. Let's see a simple example:

```
10 DEFINT A-Z: 'important for the data type to be int
20 a=5
30 print @a:'prints the memory address where a is stored
40 poke @a,7: 'this is the same as doing a=7
50 PRINT a : ' this prints a 7
```

There is a small "error" in the program. It is the POKE @a,7 instruction because it only stores one byte and the variable "a" has 2 bytes because it is an integer. This case works because the most significant byte of "a" is zero, but if "a" had a value greater than 255 then its most significant byte would not be zero and the POKE would only be altering the least significant byte. An 8BP POKE would always work because it is 16 bit:

| **I POKE,@a,7: 'put a 7 in the variable "a".**

```
a=1000
Ready
print a
1000
Ready
print Ca
432
Ready
poke Ca,7
Ready
print a
775
Ready
```

Knowing this, we only have to pass to C the address where our BASIC variable is stored. For this we have the instruction |POKE of 8BP that works with 16 bits (keep in mind that a memory address occupies 16 bits). We are going to pass the address of the variable "a" at address 40000, although we could use any other address that is free.

**|POKE, 40000, @a:**'we leave @a at address 40000

An alternative method in BASIC is to use two POKEs:

**dir=@a:**

**POKE 40001, INT (dir/256) POKE  
40000, INT (dir MOD 256)**

What we have written at address 40000 is the memory address where the variable a is stored.

**IMPORTANT:** BASIC can relocate a variable when new variables are created. This means that if you pass a C routine a memory address via an |POKE and subsequently create new BASIC variables, the memory address of the variable you have passed may have changed. **It is only guaranteed not to change if no new variables are created.** The following example illustrates this very well, there are 2 location changes

```
10 DIM a(100): i=0
20 PRINT @a(0)
30 b=2:'new variable relocates a()
40 PRINT @a(0)
50 c=2:'new variable relocates a()
60 PRINT @a(0)
70 goto 20
```

The solution to this problem is to declare all variables at the beginning of the program.

```
10 dim a(100)
20 b=2
30 c=3
40 print @a(0)
70 goto 40
```

Now from C we can access the variable "a" in two ways, although the second way (by means of a "mapped" C variable) is the most interesting.

```
// global variables
int dir_a;
int data;

int main()
{
//let's store in dir_a the address of the variable a
_8BP_peek_2(40000, &dir_a);
_8BP_peek_2(dir_a, &data); //this puts the value of the BASIC variable
// "a" into the C variable "data". This is a way to read the value of "a".
_8BP_poke_2(dir_a,7); //this puts a 7 into the BASIC variable "a".
return 0;
}
```

Let's look at the same example in another way, with a C variable that is "mapped" to the basic variable. For this we must use the notation with "\*".

```
// global variables
int dir;
int *a;

int main()
{
//let's store in dir_a the address of the variable a
_8BP_peek_2(40000, &dir);
a=dir; //to avoid compile warning use a=(int*)dir
*a=5; //this puts a 5 in the BASIC variable "a".
return 0;
}
```

We have seen how it works with simple variables. Now we are going to see how BASIC arrays work in order to access them from C. The first thing we are going to do is to understand how BASIC stores array data in memory.

```
1 DEFINT a-z
2 MODE 2
10 a=5
20 PRINT "the dir of a is @a";@a
25 PRINT "-----"
30 DIM b(5)
40 FOR i=0 TO 5
50 PRINT "the dir of b(";i;")"
is ";@b(i)
60 NEXT
70 PRINT "-----"
80 DIM c(3,4)
90 FOR j=0 TO 4:FOR i=0 TO 3
100 PRINT "the dir of c(";i;",";j;")"
is ";@c(i,j)
110 NEXT:NEXT

120 |POKE,40000,@c(0,0)
```

**With line 120 we have stored at address 40000, the memory address where the first data of the two-dimensional array is stored.**

**We could store that of the one-dimensional array with |POKE,40000, @b(0)**

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	

The above program teaches us how BASIC stores variables.

- **One-dimensional array data** is stored all in a row. Each data occupies 2 bytes because we are working with integers.

**For example, in BASIC you type:**

```
DIM b(20)
|POKE,40000,@b
```

**CALL <routine C>:'address where main() has been assembled**

```
PRINT b(8)
```

and from C you write:

```
int dir;
int *b; //variable with which we are going to access the array
BASIC int main(){
    _8BP_peek_2(40000, &dir);
    b= dir; //b is a pointer and *b[] are variables
    *b[8]=5; //add a 5 to BASIC variable b(8) return
    0;
}
```

- The **data of the two-dimensional arrays** are stored consecutively, with the variations of the first dimension producing consecutive data. In the example with **DIM(3,4)** the data (2,3) follows the (1,3) but the data (2,3) is  $4 \times 2 = 8$  bytes further than the data (2,2), because the first dimension of the array is 4. **A DIM (3,4) is an array of dimensions (4, 5) because the zero also counts.** You can check this by printing @c(3,2) and @c(2,2), you will see that there is a difference of 8.

To access the data from C it can be done with a simple pointer taking into account the first dimension when accessing. In the following example I have created an array c(12,16) and to access from C to the position (2,7) I use  $2+7*13$ , since the first dimension is  $12+1 = 13$

In BASIC you write:

```
DIM c(12,16)
|POKE, 40000, @c
CALL <routine C>;' address assembly routine
PRINT c(2,7)
```

and from c you write:

```
int dir;
int *b; //variable with which we are going to access the array
BASIC int main(){
    _8BP_peek_2(40000, &dir); //read dir 40000 and write to dir c=
    dir; //c is a pointer and *c[] are variables c[2+7*13]=5; //add a 5
    to BASIC variable c(2,7) return 0;
}
```

If you are a C programmer, you will know that in C there are double pointers that are expressed with a double asterisk (e.g. `**c`). A priori they seem suitable because you could reference data with `c[x][y]`. However, they only serve to store reserved memory dynamically with "malloc" and "calloc" instructions and require memory for both the data and the pointers in each row. This means that you could use them, but you would have to reserve memory for the pointers. I don't recommend them.

### 20.2.3 BASIC and C text strings

You should know that a string in C is a simple pointer to char, while a string variable in BASIC (for example `mivar$="hello"`) consists of a 3-byte descriptor containing the memory address where the string is located and the length of the string. That is, **a C string and a BASIC string are different things.**

So, if you want to print a variable containing a text string, you cannot pass it from BASIC to C because they are not the same thing. But you can print text strings without problems if instead of passing them from BASIC, you define them in C, for example:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stdio.h>
#include "8BP.h"
#include "minibasic.h"
#include "8BP.h"
#include "minibasic.h"

//----globalvariables
char* cad; //could be initialised here

//----functions----
int main(){
    cad="you have failed in your mission"; //we initialise it with a phrase
    _basic_print(cad); //print BASIC style
    _8BP_printat4(0,0,60,cad); //print 8BP style printat return 0;
}
```

### 20.3 Third step: Compile using "compila.bat".

We are about to take the big step: compiling to generate an AMSTRAD binary. For this step you have a help .bat, called "compila.bat". **Before taking this step you must have the SDCC compiler installed on your computer, otherwise it will fail.** You can download it from <http://sdcc.sourceforge.net/>.

**Note:** On windows10, running SDCC has failed me because it is installed in "Program files" and whitespace doesn't seem to suit it. If this is your case, just install it in a directory whose name does not include whitespace.

Once SDCC is installed, open a command window (the ses.bat script does that) and type compila.bat. The screen will change colour: green if everything went well and red if there are problems. The "compila.bat" script executes the following steps:

1. Deletes the output files of the script itself (ciclo.dsk and ciclo.bin among others).
2. Invokes SDCC to compile your program
3. Translate SDCC output to binary using hex2bin tool
4. Insert the generated binary into a disk which you call cycle.dsk, using the manageDsk tool.

After running "compila.bat", you will get a red screen in the following cases:

- Syntax errors in your C program
- The file ciclo.dsk that compila.bat is going to generate is opened by winape. In this case you must connect another disk to winape so that compila.bat can recreate it.
- Compilation errors such as a library that you have tried to use and have not included, etc.

In case of C syntax errors, or other C errors or if the file ciclo.dsk that will generate compila.bat is opened by winape, you will get a red error screen like this:

```
8888  BBBB  PPPPP  
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\proyectos09"  
" -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 COMPILACION! | "  
"+-----+  
C:\proyectos\proyectos09\_personal\8BP\V40\PROYECTO_V40_clean\C>
```

*Fig. 115 Compila.bat complains that you have C errors*

In case everything went well this would be the output

```

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 produces a green screen: everything went well.*

In case you get a green screen, it means that all the steps of compila.bat have gone well ( borSDCC, you will have valuable information on the screen and in the output subdirectory you will find files you need:

- Cycle.dsk
- Cycle.map

## 20.4 Step four: check memory limits

The compila.bat script shows you on its green screen two very valuable pieces of information: "**lowest address**" and "**highest address**". These are the memory addresses where the generated binary starts and ends. The "compila.bat" script uses the address 20000 as the compile address in the SDCC invocation and if the resulting binary file is very large, it could exceed the address 24000, thus damaging the 8BP library. You must make sure that the "**highest address**" is less than 24000. If it is not lower, **you must modify the SDCC invocation in the "compila.bat" script**.

**to compile by assigning an address less than 20000.** This way your new binary and the 8BP library will not overlap. You must modify two lines of the compila.bat script. The first one is the one that invokes SDCC. It is a very long line. You have to change the parameter 20000 to the new address

```
sdcc -mz80 --verbose --code-loc 20000 --data-loc 0 --no-std-crt0 --fomit-frame-pointer --opt-code-size -l8BP_wrapper -lmini_BASIC -o output/cycle.c
```

The second line you need to modify is the one that invokes managedsk so that it is consistent with the new memory address. This is the line and as you can see, address 20000 also appears.

```
managedsk -L "output "cycle.dsk" -I "output "cycle.bin"/CYCLE.BIN/BIN/2000000 -I "output\cycle.bin"/CICLO.BIN/BIN/20000 -S "output\ciclo.dsk" -I "output\ciclo.bin"/CICLO.BIN/BIN/20000
```

Obviously if your binary starts at 20000, your BASIC program will have to incorporate a 19999 MEMORY. This will subtract 4KB from your free space but you will also save the BASIC lines corresponding to the game cycle, so one thing for another and it's as if you haven't lost anything.

If the **highest address** is less than 24000, **you should adjust it as much as possible, i.e. as close to 24000 as possible, so as not to waste memory.** This may involve (for example) using address 21000 when invoking SDCC. Do this in order to have as much memory for BASIC as possible. You should put a MEMORY consistent with that address in your BASIC program. For example, if the binary starts at 21000, you will have to set a MEMORY of 20999.

**Ultimately, you may have to modify two lines in the "compila.bat" script to adjust the compile start address, which I initially set to 20000.**

## **20.5 Step 5: Locate the address of the function to be invoked from BASIC.**

After compiling with "compila.bat", in the subdirectory "output" you will have obtained a file called ciclo.map. In this file you must find the memory address of the function or functions you intend to invoke from BASIC. In this example we are only going to invoke the function main(), which is located at &56b0 as can be seen in this fragment of the file ciclo.map

00005699	__basic_paper
0000566B	__basic_plot
00005682	__basic_move
00005699	__basic_draw
000056B0	_main
00005920	_abs
0000592C	_strlen
0000593B	__modchar
00005948	__modint
00005954	__moduchar

*Fig. 117 the address of each function is in ciclo.map*

This means that to invoke the main() function from BASIC we simply do:

**CALL &56B0**

Be careful because if you make any changes in the compilation phase (modification of the .c-cycle or changes in the memory addresses of the compila.bat script) the address of each function may change when re-compiling.

## **20.6 Step 6: Include the new binary in your .dsk game**

You have already generated a cycle.dsk file containing the cycle.bin file that you must load in order to invoke the main function (and/or any other functions you want). To get both this file and your game on the same disk, you must select cycle.dsk from winape and simply load the cycle.bin file.

**LOAD "CYCLE.BIN",20000**

Then from the winape menu you select your disc (where you have your game) and save this binary

**SAVE "CICLO.BIN", b, 20000, <length>.**

where length =highest address - lowest address +1

Now in your loader.bas file you must load this new additional binary and invoke it from your BASIC listing with CALL <address>.

```
10 MEMORY 24999
15 LOAD "!paint.scr",&c000: 'only if your game has loading screen
20 LOAD "yourgame.bin"
25 LOAD "cycle.bin", 20000
50 RUN "yourgame.bas"
```

That's all. You can now program in C with 8BP!

## 20.7 8BP function reference in 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 routed>, <flag routed>, <flag routed>, <flag routed>.	void _8BP_autoall(); void _8BP_autoall_1(int flag);
COLAY, threshold_ascii, @collision, #  COLAY, @collision, #  COLAY, #  COLAY	void _8BP_colay_3(int threshold, int* collision, int sp); void _8BP_colay_2(int* collision, int sp); void _8BP_colay_1(int sp); void _8BP_colay();
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%.  COLSP, 32, ini, end  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* collision); void _8BP_colsp_1(int sp);
COLSPALL,@who%,@who%,@with whom%  COLSPALL, collider  COLSPALL	void _8BP_colspall_2(int* collider, int* collided); void _8BP_colspall_1(int collider_ini); void _8BP_colspall();
LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$, @string\$, @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, song, speed  MUSIC, flag, song, 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* data);
POKE, dir, value	void _8BP_poke_2(int address, int data);
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,colour1,colour2,...,colourN  RINK, jump	void _8BP_rink_N(int num_params,int* ink_list); void _8BP_rink_1(int step);
ROUTEESP, #, steps	void _8BP_routesp_2(int sp, int steps); 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, value  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, colour, dy, dx	void _8BP_stars_5(int star_ini, int num_stars,int colour, 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);

Here are some examples of use, which complement the example used in the BASIC to C translation.

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,@who%,@who%,@withwhom%</b>  10 collider%=0: collided%=0 20  COLSPALL, @collider%, @colliderd%, @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,colour1,colour2,...,colourN</b> <b> RINK, jump</b>  10  RINK,1,2,2,2,3,3 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 inks[5]={1,2,2,3,3}; <b>_8BP_rink_N(5,inks);</b> <b>_8BP_rink_1(1);</b>

RSX	<b> LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$, @string\$, @string\$.</b>  10 cad\$="YYYYYY YYYYYY YYYYYY YYYYYY" 20  LAYOUT,10,1,@cad\$
C	<b>void _8BP_layout_3(int y, int x, char* cad);</b>  <b>_8BP_layout_3(10,1," YYYYYY YYYYYY YYYYY YYYYY";</b>  Or:  <b>char* cad=" AA YYYYYY YYYYY YYYYY YYYYY";</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\$ 20  PRINTAT,0,100,40,@cad\$ 20
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 BASIC function reference in C ("minibasic")

This is the small set of BASIC-like instructions that 8BP provides you with through the minibasic.h library, to easily translate your BASIC listing to C. If you are trying to translate only the game cycle, this set of instructions will suffice.

BASIC	C prototype
BORDER	<code>void _basic_border(char colour);</code> <code>//example _basic_border(7)</code>
CALL	<code>void _basic_call(unsigned int address);</code> <code>// 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);</code> <code>//takes around 0.3 ms. slow but simple</code>
LOCATE	<code>void _basic_locate(unsigned int x, unsigned int y);</code> <code>// 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);</code> <code>//example: _basic_print("Hello")</code>
RND	<code>unsigned int _basic_rnd(int max);</code> <code>//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);</code> <code>//similar to STR\$</code> <code>//example: _basic_print(_basic_str(num))</code>
TIME	<code>unsigned int _basic_time();</code> <code>//return an unsigned int,(0..65535). As integer, when</code> <code>// reach 32768 go to -32768</code>



# 21 8BP Library Reference Guide

## 21.1 Library functions

### 21.1.1 |3D

This command activates 3D projection in the PRINTSP and PRINTSPALL commands  
To project we have the command |3D

Use

To activate 3D projection:

**|3D, 1, <Sprite\_fin>, <offsety>, <offsety>, <offsety>, <offsety>, <offsety>.**

To deactivate it:

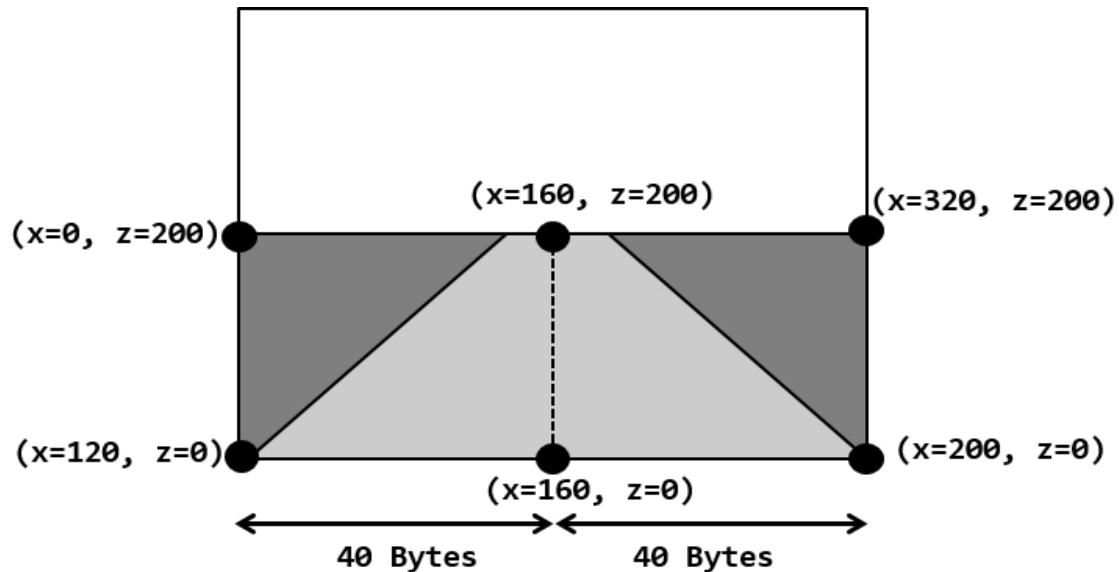
**|3D, 0**

The affected sprites are from Sprite 0 to <Sprite\_fin>. This command activates 3D projection in the **|PRINTSP** command and in **|PRINTSPALL**. This means that before printing to the screen, the "projected" coordinates will be calculated and then printed to the screen. The coordinates of the sprites are not affected, i.e. the 2D coordinates in the sprite table will remain the same.

This command **does not affect collision mechanisms**, i.e. if we use COLSPALL and detect a collision between projected sprites, the collision is occurring in the 2D plane.

As for the last parameter <offsety> it is to project higher or lower, so that we can place the game markers where we want. When projecting the screen, which is 200 pixels high, it becomes 100 pixels high, so we can choose how high we place the projection. If a Sprite is not affected by the projection because it is higher than <Sprite\_fin>, then it is not affected by <offsety> either.

The following figure represents which world map coordinates are projected at certain representative points on the screen when **|MAP2SP** is invoked with (yo=0, xo=0).



*Fig. 118 projected world coordinates*

If instead of  $(xo=0, yo=0)$  we use another coordinate for MAP2SP, the 2D world coordinates corresponding to the points referenced in the image will be shifted by  $(x, z)$  as indicated by  $xo$  and  $yo$ .

### 21.1.2 |ANIMA

This command changes the animation frame of a sprite, taking into account its assigned animation sequence.

Use:

**|ANIMA, <sprite number>, <sprite number>, <sprite number>, <sprite number>.**

Example:

**|ANIMA,3**

The command queries the animation sequence of the sprite, and if it is non-zero then it goes to the animation sequence table (the first valid sequence is 1 and the last one is 31). It chooses the image whose position is next to the current frame and updates the frame field of the sprite attribute table.

If the next frame in the sequence is zero then it is cycled, i.e. the first frame of the sequence is chosen.

In addition to changing the frame field, the image field is changed and the memory address of where the new frame is stored is assigned.

|ANIMA does not print the sprite, but leaves it ready for printing, so that the next frame of its sequence is printed.

|ANIMA does not check that the animation flag is active in the sprite's status byte. In fact, our character will normally only want to be animated when it moves and not always when it is printed.

If the animation sequence is a "death sequence" (includes a "1" in its last frame), then upon reaching the frame whose image memory address is 1, the sprite will become inactive.

The 8BP library allows you to make "death sequences", which are sequences that, upon completion, the sprite goes to an inactive state. This is indicated by a simple "1" as the value of the memory address of the final frame. These sequences are very useful for defining explosions of enemies that are animated with |ANIMA or |ANIMALL. After hitting them with your shot, you can associate a death animation sequence to them and in the following game cycles they will go through the different animation phases of the explosion, and when they reach the last one they will go to inactive state, not being printed any more. This inactive state is done automatically, so what you have to do is simply check the collision of your shot with the enemies and if it collides with any of them you change the state with |SETUPSP so that it cannot collide any more and you assign it the animation sequence of death, also with |SETUPSP.

If you use a death sequence, don't forget to make sure that the last frame before you find the "1" is a completely empty one, so that there is no trace of the explosion.

Example of a death sequence

```
dw EXPLOSION_1,EXPLOSION_2,ExPLOSION_3,1,0,0,0,0,0
```

### 21.1.3      |ANIMALL

This command animates all sprites that have the animation flag set in the status byte.  
This command has no parameters

Use

**|ANIMALL**

**IMPORTANT:** since version v37 of the library, this command is only accessible via CALL (see table of correspondences in the appendix), and not via RSX command. Removing it from the list saved a few bytes of memory and it can still be used either from a parameter in PRINTSPALL or from a CALL call.

It is recommended if you are going to animate a lot of sprites as it is much faster than invoking the **|ANIMA** command several times.

As you will normally want to invoke **|ANIMALL** every game cycle, before printing the sprites, there is a more efficient way to invoke it, and that is to set the corresponding parameter of the **|PRINTSPALL** command to "1", i.e.

**|PRINTSPALL,1,0**

This function internally invokes **|ANIMALL** before printing the sprites, saving 1.17ms compared to the time it would take to invoke separately **|ANIMALL** and **|PRINTSPALL**

### **21.1.4 |AUTO**

This command moves a sprite (changes its coordinates) according to its velocity attributes Vy,Vx. These attributes are whatever the sprite has in the sprite table.

Use:

**|AUTO, <sprite number>, <sprite number>, <sprite number>, <sprite number>.**

Example:

**|AUTO, 5**

What this command does is update the coordinates in the sprite table, adding the velocity to the current coordinate.

The new coordinates are

new X = current X-coordinate + Vx

new Y = current Y-coordinate + Vy

It is not necessary for the sprite to have the automatic movement flag active in the status field.

### **21.1.5 |AUTOALL**

This command moves all sprites that have the automatic movement flag active, according to their speed attributes Vy, Vx.

Use:

**|AUTOALL, <routing flag>, <routing flag>.**

Example

**|AUTOALL,1** invokes **|ROUTEALL** before moving sprites

**|AUTOALL,0** does not invoke **|ROUTEALL**

**|AUTOALL** the last used value is used as parameter (has memory)

The routing flag is optional. Since the **|ROUTEALL** command does not modify the coordinates of the sprites, they must be moved with **|AUTOALL** and printed (and animated) with **|PRINTSPALL**. That's why you have an optional parameter in **|AUTOALL**, so that **|AUTOALL,1** internally invokes **|ROUTEALL** before moving the sprite, saving you a BASIC invocation that will always take a precious millisecond.

### **21.1.6 |COLAY**

Detects the collision of a sprite with the screen map (the layout). It takes into account the size of the sprite to see if it collides, and considers that the layout elements are all 8x8 pixels of mode 0 (i.e. 4 bytes x 8 lines). It can be invoked with 3,2,1 or no parameters. If invoked without parameters, the values of the last call with parameters will be used and it is much faster.

Use:

```
|COLAY, <ASCII threshold>, @collision%, <	sprite number>, <sprite number>.  
|COLAY, @collision%, <sprite number>  
|COLAY, <num_sprite>, <num_sprite>.  
|COLAY
```

The optional parameter <ASCII threshold> need only be used in a first invocation to set the collision threshold in the **|COLAY** command. This threshold represents the largest ASCII code of the layout element that is considered as "no collision". The default is 32 (the white space). To set the threshold you want, refer to the ASCII table of the **|LAYOUT** command.

Example:

```
|COLAY, 65, @col%,31 : rem sprite is 31, threshold is 65
```

The variable you use for collision can be called whatever you want. I have put "col".

This routine modifies the collision variable (which must be an integer, hence the "%") by setting it to 1 if there is a collision of the indicated sprite with the layout. If there is no collision, the result is 0.

```
10 xprevious=x  
20 x=x+1  
30 |LOCATESP,0,y,x: ' position sprite in new position  
40 |COLAY,@collision%,0: 'collision check
```

Now we check the collision and if there is a collision we leave it in its previous location.

```
50 if collision%=1 then x=xprevious: LOCATESP,0,y,x
```

You can also use the **|MOVER** command to position the sprite and do the check.

```
10 |COLAY,65,@col,31: 'configuration. We only do it once  
20 |MOVER,31,1,1: ' we move it to the right and downwards  
30 |COLAY: ' invoke it without parameters (faster)  
30 if col THEN MOVER,31,-1,-1 : rem has collided and so I leave it where it was.
```

## 21.1.7 |COLSP

This command allows to detect the collision of a sprite with the other sprites that have the collision flag active.

Usage :

To configure:

```
|COLSP, 32, <sprite initial>, <sprite final>.  
|COLSP, 33, @collision%  
|COLSP, 34, dy, dx
```

For collision detection:

**|COLSP,<sprite number>, @colsp%.**

Example:

**col%=0**

**|COLSP,0,@col%**

The function returns in the variable that we pass as parameter, the number of the sprite with which it collides, or if there is no collision it returns a 32 because the sprite 32 does not exist (there are only from 0 to 31).

**IMPORTANT:** The collision variable in the COLSP command is not the one used in the COLSPALL command. They are different variables (unless you pass both commands the same variable to act on it).

Like sprite printing with PRINTSPALL, the COLSP function checks sprites starting at 31 and ending at zero. If they have an active sprite collision flag (bit 2 of the status byte) then the collision is checked. If two sprites collide at the same time with our sprite, the larger sprite number is returned as it is the one that is checked first.

#### **Invocations to configure the command:**

There is a way to configure COLSP to do less work by collision checking fewer sprites to save execution time. The configuration will be indicated by the use of sprite 32 (which does not exist).

**|COLSP, 32, <sprite initial to check>, <sprite final to check>.**

If for example our character's enemies are sprites 25 to 30, and we configure them as colliders (not colliders) we can invoke (only once) the command like this:

**|COLSP, 32, 25, 30**

This means that any subsequent invocation of the |COLSP command should only check the collision of sprites 25 to 30 (as long as they have the "collided" flag active).

For example, if we only have to check 6 enemies, by pre-configuring the command to only check from 25 onwards, we can save up to 2.5ms on each execution. This becomes especially important in games where the character can shoot, since in each game cycle at least the collision of the character and the shots will have to be checked.

Another interesting optimisation, capable of saving 1.1 milliseconds in each invocation, is to tell the command to always use the same BASIC variable to leave the result of the collision. To do this, we will indicate it using 33 as the sprite, which also does not exist.

**col%=0**

**|COLSP, 33, @col%, @col%, @COLSP, 33, @col%, @COLSP, 33, @col%**

Once these two lines have been executed, subsequent invocations of COLSP will leave the result in the variable col, without the need to indicate it, for example:

### |COLSP, 23

Finally, it is possible to adjust the sensitivity of the COLSP command, deciding whether the overlap between sprites must be of several pixels or only one, in order to consider that a collision has occurred.

This can be done by setting the required number of overlapping pixels in both the Y and X directions, using the COLSP command and specifying sprite 34 (which does not exist).

### |COLSP, 34, dy, dx

The default values for dy and dx are 2 and 1 respectively. Note that in the y-direction they are considered pixels, but in the x-direction they are considered bytes (one byte is two pixels in mode 0).

For a detection with a minimum overlap (one pixel vertically and/or one byte horizontally) you must do:

### |COLSP, 34, 0, 0

## 21.1.8 |COLSPALL

Use:

To configure:

```
|COLSPALL,@collider%, @collider%, @collider%, @collider%,
@collider%, @collider%.
```

To check for collisions

```
|COLSPALL
|COLSPALL, <initial collider>.
```

This function checks who has collided (among the group of sprites that have the collider flag of the status byte set to "1") and with whom it has collided (among the group of sprites that have the collider flag of the status byte set to "1").

This is a highly recommended feature when you have to handle collisions of your character and multiple shots, as it saves invocations of |COLSP and therefore speeds up your game.

**Important:** the colliders (status bit 5) are checked from 31 to 0. For each collider, the colliderables (status bit 1) are also checked from 31 to 0.

In case COLSPALL is invoked with a single parameter,

**|COLSPALL, <initial collider>.**

Colliders will be scanned from the indicated collider -1 to sprite zero, in descending order. This way if you need to detect more than one collision per cycle of

game, you can do this by successively invoking **COLSPALL, <collider>** until the variable collider takes the value 32

**Example:**

**|COLSPALL, 7 : rem searches for collisions from collider 6**

### 21.1.9 |LAYOUT

Use:

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

Example:

```
string$ = "XYZZZZ      ZZ"  
|LAYOUT, 0,1, @string$
```

Note that using **|LAYOUT, 0,1, "XYZZZZZZ ZZ"** would be incorrect on a CPC464 although it works on a CPC6128. Also, on CPC6128 you can skip the use of the "@" but on CPC464 it is mandatory.

This routine prints a row of sprites to build the layout or "maze" for each screen. In addition to drawing the maze, or any on-screen graphics built with small 8x8 sprites, you can also detect collisions of a sprite with the layout, using the **|COLAY** command.

The sprites to be printed are defined with a string, whose characters (32 possible) represent one of the sprites following this simple rule, where the only exception is the blank space representing the absence of a sprite.

Character	Sprite id	ASCII code
" "	NONE	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
"T"	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

**Table 6 Character and Sprite mapping for the |LAYOUT command**

**IMPORTANT:** After printing the layout you can change the sprites to be characters, so you will still have the 32 sprites.

The y,x coordinates are passed in character format. The library internally maintains a 20x25 character map, so the coordinates take the following values:

y takes values [0,24]

x takes values [0,19].

The sprites to be printed must be 8x8 pixels. they are "bricks", also called "tiles", and are often used in the same way.

If you use other sprite sizes, this function will not work well. It will actually print the sprites, but if a sprite is large you will have to place blanks to make room for it.

The library maintains an internal layout map and this function updates the internal layout map data so that it will be possible to detect collisions. This map is an array of 20x25 characters, where each character corresponds to a sprite.

The @string is a string variable. You cannot pass the string directly, although in the CPC6128 the parameter passing allows it, but it would be incompatible with CPC464.

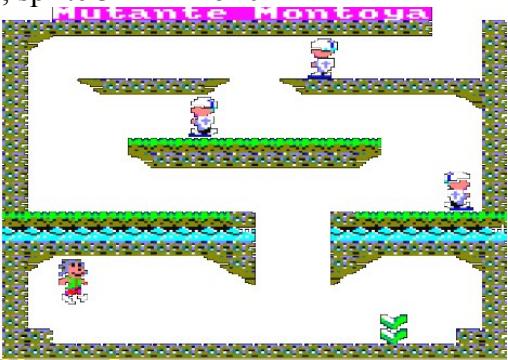
#### Precautions:

The function does not validate the string you pass it. If it contains lowercase letters or any other character different from the allowed ones, it may cause undesired effects, such as a computer restart or crash. It can't be an empty string either!

The limits set with SETLIMITS should allow you to print where you want. If you later want to clipping in a smaller area you can invoke SETLIMITS again when the whole layout is printed.

Example:

<pre>2070  SETLIMITS,0,80,0,200 2090 c\$(1)= " PPP PPPPP 2100 c\$(2)= "PU 2110 c\$(3)= "P 2120 c\$(4)= "P"</pre>	<p>This example uses several bricks that have been previously created with the "SPEDIT" tool. ; sprite 20 --&gt; O bush</p>
--	---

<pre> 2130 c\$(5)= "P      TPPPPPPPU                   TPPPPPPPPPPPPPPP" 2140 c\$(6)= "P      TP" 2150 c\$(7)= "P      P" 2160 c\$(8)= "P      P" 2170 c\$(9)= "P      YYYYYYYYYYYYYYYY                   P" 2190 c\$(10)="P     TPPPPPPPPPPPPPU                   P" 2195 c\$(11)="P      P" 2200 c\$(12)="P      P" 2210 c\$(13)="P      P" 2220 c\$(14)="YYYYYYYYYYYYYYYYYP                   PYYYYYYYYYYY" " 2230 c\$(15)="RRRRRRRRRRRRRRR                   RRRRRRRR" 2240 c\$(16)="PPPPPPPPPPPPPPP                   PPPPPPPPPP P" 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)="PPPPPPPPPPPPPPPPPPPPPPPPPPPPPP PPPPP". 2280 for i=0 to 23 2281  LAYOUT,i,0,@c\$(i) 2282 next </pre>	; sprite 21 --> P rock ; sprite 22 --> Q cloud ; sprite 23 --> R water ; sprite 24 --> S window ; sprite 25 --> T arch of right bridge ; sprite 26 --> U arch of left bridge ; sprite 27 --> V flag ; sprite 28 --> W plant ; sprite 29 --> X tower spike ; sprite 30 --> And grass ; sprite 31 --> Z brick 
--	--

### 21.1.10 |LOCATESP

This command changes the coordinates of a sprite in the sprite attribute table.

Use

**|LOCATESP, <sprite number>, <y>, <x>**

Example

**|LOCATESP,0,10,20**

An alternative to this command, if we only want to change one coordinate, is to use the BASIC POKE command, inserting the value we want in the memory address occupied by the X or Y coordinate. If we want to enter a negative coordinate, the **|POKE** command is necessary, since it would be illegal with the BASIC POKE command.

The **|LOCATE** command does not print the sprite, it only positions it for when it is printed.

### 21.1.11 |MAP2SP

This function traverses the world map described in the `map_table.asm` file and transforms the map items that may be partially or completely entering the screen into sprites.

Use

|MAP2SP, <y>, <x>

|MAP2SP, <status>, <status>, <status>, <status>, <status>, <status>.

Example

|MAP2SP, 1500, 2500

The sprites created by MAP2SP are created by default with state 3, i.e. with the print flag active (**|PRINTSPALL** prints it) and with the collision flag active (**|COLSP** will collide with it). If you need the sprites to be created with another state, you simply invoke the **|MAP2SP** command once with a single parameter indicating the state with which the sprites should be created.

If by any chance **|MAP2SP** encounters more than 32 items to translate into sprites, it will ignore those exceeding 32.

**|MAP2SP, <status>, <status>, <status>, <status>, <status>, <status>.**

**|This configures the MAP2SP command to be printed but not collisional.**

**IMPORTANT:** On the world map you can combine normal images with "background images" (section 8.5). Background images always have transparency. The transparency flag you use in **|MAP2SP, <status>** will only apply to normal images.

The **<y>,<x>** parameters of the function are the moving origin from which the world is displayed on the screen. There are three other parameters found in the **MAP\_TABLE**, the table from which the world is defined. These parameters are the maximum height, the maximum width (in negative) and the number of items in the world (maximum 82).

Each item is a tuple of 3 parameters preceded by the mnemonic "DW":

**DW Y, X, <image>**

**MAP TABLE**

;

**3 parameters before the list of "map items".**

**dw 50 ; maximum height of a sprite in case it gets through the top  
and part of it has to be painted.**

**dw -40 ; maximum width of a sprite in case of a left-handed sprite  
(negative number)**

**db 64 ; number of map items to be considered. at the most it should be 82**

**and from here start the items dw**

**100,10,HOUSE; 1**

**dw 50,-10,CACTUS;2**

**dw 210,0,HOUSE;3**

**dw 200,20,CACTUS;4**

**dw 100,40,HOUSE;5**

**dw 160,60,HOUSE;6**

**dw 70,70,HOUSE;7**

**dw 175,40,CACTUS;8**

**dw 10,50,HOUSE;9**

**dw**

**250,50,HOUSE;10**

**dw**

**260,70,HOUSE;11**

**dw**

**260,70,HOUSE;11**

**dw 290,60,CACTUS;12**

**dw 180,90,HOUSE;13**

**dw 60,100,HOUSE;14**

**dw 60,100,HOUSE;14**

**...**

### **21.1.12 |MOVER**

This command moves a sprite relatively, i.e. by adding relative quantities to its coordinates.

Use:

**|MOVER,<sprite number>, <dy>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>.**

Example:

**|MOVER,0,1,-1**

The example moves sprite 0 down and to the right at the same time. The sprite does not need to have the relative movement flag enabled.

There is a way to use **|MOVER** without specifying either "dy" or "dx". To do this, we will specify sprite 32, which does not exist, and we will put as parameters the memory addresses of the variables we want to use to store both "dy" and "dx".

The memory address of a variable is obtained by simply prefixing it with the symbol "@".

Example:

**dy%= 5**

**dx%= 2**

**|MOVER,32, @dy, @dx**

From this moment on, we will be able to use:

**|MOVER, <id>**

And with that the sprite "id" will move as indicated by the variables dy, dx. This mechanism also works with **|MOVERALL**

### **21.1.13 |MOVERALL**

This command moves relatively all sprites that have the relative movement flag activated.

Use

**|MOVERALL, <dy>, <dx>, <dx>, <dx>, <dx>, <dx>.**

Example

**|MOVERALL,2,1**

The example moves all sprites with relative movement flag down (2 lines) and 1 byte to the right.

If no parameters are specified, the variables specified in the MOVER invocation with sprite 32 will be used, i.e.

**|MOVER,32, @dy, @dx**

**|MOVERALL**

Equivalent to **|MOVERALL, dy, dx**

This "advanced" use of the command avoids the passing of parameters in each invocation and is therefore faster, which is essential in our BASIC programs.

### 21.1.14 |MUSIC

This command allows a melody to start playing Usage:

|MUSIC,<flag\_channel\_C>,<flag\_repeat>,<melody\_number>,<speed>.   
|MUSIC,<flag\_repetition>,<melody\_number>,<speed>.

#### |MUSIC : without parameters music ends playing

The C-channel flag with value 1 allows the third sound channel to be left free so that it can be used to make sound effects (triggers etc) with the command

SOUND 4,<note>,<duration>, ...

Note that you must use channel 4, as in BASIC the channels are typed as A=1, B=2, C=4.

If the channel flag is omitted or set to zero, then the music will use all 3 channels and you will not be able to use the SOUND command at the same time (you can but with weird effects).

The repeat flag must be zero to loop the music. If you only want the music to play once, you should use the value 1.

The melody number shall be between 0 and 7.

The "normal" speed is 6. If we use a higher number it will play slower and if the number is lower it will play faster.

The |MUSIC command invoked without parameters disables music interruption and stops playing any melody.

Examples:

```
|MUSIC,0,0,0,0,6  
|MUSIC,1,0,0,0,6  
|MUSIC,0,0,6  
|MUSIC
```

Internally, what the command does is to install an interrupt that is triggered 300 times per second. If we set speed 6, one out of every 6 times it is triggered, the music playback function is executed.

Because it is interrupt-based, there needs to be a program running for the music to play, because while the BASIC interpreter is waiting for commands, such interrupts are not enabled. If you simply run the |MUSIC command, you won't hear anything, but if you run it inside a program like the one shown below, the music will play.

<b>10  MUSIC,0,0,0,5 20 goto 20: ' infinite loop. When running, the music plays</b>
---

### 21.1.15 |PEEK

This command reads a 16-bit data value from a given memory address. It is intended for querying the coordinates of sprites that move with automatic or relative motion.

Use

**|PEEK,<address>, @data%.**

Example

**data%=0**

**|PEEK, 27001, @dato%, @dato%, @dato%, @dato%, @dato%, @pEEK**

If the coordinates are only positive and less than 255 you can use the BASIC PEEK command, as it is somewhat faster.

### **21.1.16 |POKE**

This command inserts a 16-bit data (positive or negative) into a memory address. It is intended for modifying sprite coordinates, as the POKE command cannot handle negative coordinates or coordinates greater than 255 since POKE works with bytes while **|POKE** is a 16bit command.

Use:

**|POKE, <address>, <value>.**

Example:

**|POKE, 27003, 23**

This example puts the value 23 at the x-coordinate of sprite 0.

It is a very fast function, although if you are going to handle only positive coordinates it is better to use POKE as it is even faster.

### **21.1.17 |PRINTAT**

**|PRINTAT** can print a string of characters using a new, smaller character set (I call them "mini-characters"). This new command allows you to use the transparency mechanism of the sprites, so you can print characters while respecting the background. It works as follows:

Use:

**|PRINTAT,<flag transparency>, y,x,@string**

Example:

**cad\$= "Hello".**

**|PRINTAT,0,100,10, @cad\$**

The **|PRINTAT** command prints character strings and not numeric variables, so if you want to print a number (for example, the points on the scoreboard in your video game) you must do so:

```
points=points+1
cad$= str$(points)
|PRINTAT,0,100,10, @cad$
```

The |PRINTAT command is not affected by the limits for clipping set with |SETLIMITS. This is most logical since you will normally use PRINTAT to print scores on your markers, which will be outside the area bounded by |SETLIMITS.

Unlike the BASIC PRINT command, the |PRINTAT command is quite fast and can be used to update your game markers frequently.

PRINTAT uses a redefined alphabet, which may contain a reduced or different version of the "official" Amstrad characters. By default 8BP provides a small alphabet composed of numbers, capital letters and some symbols. It is as follows:

**"0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ!: ,."**

You will not be able to use a character that is not in this set, such as lower case letters. If you try to do so, the last character defined in the string (in this case the space) will be printed.

The characters of this alphabet are all the same size: 4 pixels wide x 5 pixels high, i.e. 2 bytes x 5 lines.

The default alphabet does not contain lowercase letters and many symbols are missing, although you can create your own alphabet that contains them.

### **21.1.18 |PRINTSP**

Use:

**|PRINTSP, <sprite id >, <y >,<x>, <x>.**  
**|PRINTSP, <sprite id >**  
**|PRINTSP, 32, <number of background pixels>.**

Example:

Print sprite 23 at coordinates y=100, x=40 (updating its coordinates).

**|PRINTSP, 23,100,40**

Example:

Prints sprite 23 at the coordinates already assigned in the sprite table:

**|PRINTSP, 23**

If sprite 32 (which does not exist) is specified, then the following parameter is used to specify the number of background bits in the transparent print. If 1 bit is used, then 2 background colours can be used. If 2 is specified, then 4 background colours can be used.

If a sprite\_id <32 is specified then the command prints a sprite on the screen, and if coordinates are specified, it also updates them.

The coordinates considered are:

- Number of lines in vertical [-32768..32768]. The corresponding lines inside the screen are [0..199].
- Number of bytes in horizontal [-32768..32768]. Those corresponding to the inside of the screen are [0..79].

Normally in video game logic you will make use of **|PRINTSPALL**, as it is quicker to print them all at once. However, at other times in the game you may want to print sprites separately. This example shows the lowering of a "curtain", using a single sprite that repeats horizontally and as it descends it "tints" the screen red, giving the sensation of a curtain descending.

```

7089 telon=&8ec0
7090 |setupsp,1,9,curtain
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 An example of the use of PRINTSP*

### 21.1.19 |PRINTSPALL

This routine prints all sprites with the status bit0 set at once.

Use:

**|PRINTSPALL, <ordenini>, <ordenfin>, <flag anima>, <flag sync>.**  
**|PRINTSPALL, <order type>,**

Example:

With the following values, the command prints all sprites animating them first and unsynchronised with sweep, and without sorting:

**|PRINTSPALL, 0, 0, 0, 1, 0**  
**|PRINTSPALL, 0, 1, 0: rem if commandini is omitted, takes the last value assigned, or zero if never assigned**

**animation flag**, can be set to 1 or 0. If 1 is set, then before printing the sprites the frame is changed in its animation sequence, as long as the sprites have the status bit 3 set.  
**IMPORTANT:** Animation is done before printing, not after printing. This means that if you have just assigned an animation sequence, you will not see the first frame of that sequence.

The **<flag sync>** is a synchronisation flag with the screen scan. It can be 1 or 0 . Synchronisation only makes sense if you compile the program with a compiler like "Fabacom". BASIC logic runs slowly and synchronising with the sweep produces small additional waits in each cycle of the game so it is not convenient.

As a rule of thumb, it is only suitable if your game is capable of generating 50fps per second, or in other words, a full game cycle every 20 milliseconds. If you compile your game with a compiler such as "fabacom", then it is recommended that you synchronise with the screen sweep because you will almost certainly achieve that 50fps, and you will be able to get a full game cycle every 20 milliseconds.

If you exceed them, your game will produce more frames than the screen can display and then some will not be displayed and the movement will not be smooth.

The more sprites you have on screen printing, the longer the command will take, although it is very fast. There are many sprites that may appear on screen, but do not need to be printed (they may have the 0 status bit off) such as fruits, coins, bonus elements in general and/or characters that do not move and have no animation. Even if they are not printed, they may have the collision bit set and thus affect the routine.  
|COLSP and |COLSPALL

**The order parameters ("ordenini", "ordenfin")** indicate the initial and final sprites that define the group of sprites ordered by "Y" coordinate that we are going to print. For example, if we assign the values 0,0 then they will be printed sequentially from sprite 0 to sprite 31. If we assign 0,8 they will be printed from 0 to 8 ordered (9 sprites) and from 10 to 31 sequentially. If we set 0,31 all sprites will be printed in order. If we set a 10,20 the sprites will be printed sequentially from 0 to 9, then they will be printed ordered from 10 to 20 and finally they will be printed sequentially from 21 to 31.

The ordering is very useful for making "Renegade" or "Golden AXE" type games, where it is necessary to give a depth effect.

If the "ordenini" parameter is omitted, the last assigned value is considered, or zero if no value has ever been assigned. Also, if you are going to modify either of the two sorting parameters, it is a good idea to first run PRINTSPALL,0,0,0,0,0 so that the sprites are first reordered sequentially before sorting them with a new configuration.

Printing in order is more computationally expensive than printing sequentially. If you only have 5 sprites that need to be sorted, pass a 4 as the sort parameter, don't pass a 31. Sorting all the sprites takes about 2.5ms but if you sort only 5 you can save 2ms. Maybe you have a lot of sprites and it's not worth sorting some of them, like the shots or sprites that you know won't overlap.

The sorting of **|PRINTSPALL** is partial, i.e. only one pair of unordered sprites is sorted at each invocation. You may sometimes want the sorting to be complete for each frame. That is, not to sort a pair of sprites in each invocation of **|PRINTSPALL**, but to be sure that they are all sorted. The 8BP library allows you to do this through its four sorting modes, which you can set by invoking the **|PRINTSPALL** command with a single parameter (just run it once to set the sorting mode):

<b>PRINTSPALL,0 : partial sorting using Ymin</b>	<b>PRINTSPALL,1 : complete sorting using Ymin</b>	<b>PRINTSPALL,2 : partial sorting using Ymax</b>	<b>PRINTSPALL,3 : complete sorting using Ymax</b>
--	---	--	---

Sorting using Ymax is based on the largest Y-coordinate of the sprites, i.e. where their feet are located rather than their heads. If the sprites are the same size, a Ymin-based sorting may work, but if the sprites are of different heights you may want to sort according to where each character's feet are, and for that you will have to use sorting mode 2 or 3.

Ymax sort modes are slower, about 0.128 ms per sprite, so use them when you really need them.

The full sort consumes very little more than the partial sort (about 0.3ms). This is because the sprites are hardly ever jumbled from one frame to the next, but even those 0.3ms are worth saving if possible.

There is a very interesting behaviour of this function to save 1ms in its execution. It consists of invoking it with parameters once and invoking it without parameters the following times. In that case, it will be assumed that, even if no parameters are passed, their values are equal to the last ones passed. This way the parser works less and reduces the execution time.

### **21.1.20 |RINK**

This command allows you to perform an animation by inks. Usage:

**|RINK, <initial\_ink>, <colour1>, <colour2>, . . . , <colourN>,**

**<colourN>, <colourN>, <colourN>, <colourN>, <colourN>.**

**|RINK, <step>**

RINK rotates a set of inks starting at the initial ink N inks (any number of inks), according to the size of the colour pattern that is defined.

The speed of rotation can be controlled by the step parameter, which indicates the number of colour jumps each ink makes in one invocation.

Recommendation: due to the use of interruptions **|RINK** causes "stuttering" in some cases when used at the same time as the **|MUSIC** command at speed 6. In case you want to use both at the same time without interference, use another speed for music (you can use speed 5 or 7, both will work fine).

Examples:

This command defines a pattern of 4 reds (colour =3) and 4 yellows (colour =24) to be rotated from ink 8 to ink 15.

**|RINK, 8, 3, 3, 3, 3, 24, 24, 24, 24, 24**

This command defines a pattern of 2 whites (colour =26) and 2 greys (colour=13) to be rotated from ink 3 to ink 6.

**|RINK, 3, 26, 26, 13, 13, 13**

Rotate one colour of the pattern all inks

**|RINK, 1**

In an 8-colour pattern, the following command leaves everything as it was

**|RINK, 8**

This command would do nothing except force the inks to adopt the colour of the pattern.

**|RINK, 0**

### 21.1.21 |ROUTEALL

This command allows you to route all sprites that have the route flag active in their status byte through their assigned route (parameter 15 of |SETUPSP).

Use:

**|ROUTEALL**

It has no parameters so it is very easy to invoke. What this command does internally is to keep a step count for the segment that each sprite is running, so that if the segment runs out, it alters the speed of the sprite.

The command does not modify the coordinates of the sprites, so they must be moved with |AUTOALL and printed (and animated) with |PRINTSPALL. This is why you have an optional parameter in |AUTOALL, so that |AUTOALL,1 internally invokes |AUTOALL,1.

|ROUTEALL before moving the sprite, saving you a BASIC invocation that will always take a precious millisecond.

Routes are defined in the routes file routes\_yourgame.asm

#### DEFINITION OF EACH ROUTE

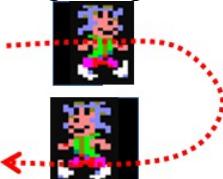
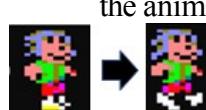
```
;=====
ROUTE0; a circle
;-----
db 5,2,0
db 5,2,-1
db 5,0,-1
db 5,-2,-1
db 5,-2,0
db 5,-2,1
db 5,0,1
db 5,2,1
db 0
```

The last segment is zero, indicating that the path is finished and the sprite must start from the beginning. Make sure that the number of steps in each segment does not exceed 250 and that both Vy and Vx are between -127 and 127.

To assign a path to a sprite you must use the SETUPSP command by specifying parameter 15. The following example associates path 3 with sprite 31

**|SETUPSP, 31, 15, 3**

There are 4 functionalities that you can use in the middle (**not at the end**) of any route:

Escape code (field "number of steps")	Description	Example
255	Change of sprite state.	DB 255, 3, 0 State goes to value 3. The zero at the end is a filler.
254	Change of sprite animation sequence  After changing the sequence, if you want the image to change as well, you must use the code 251	DB 254, 10, 0 The sequence 10 is associated. The zero is a padding. If the assigned sequence is the one the sprite already has, then it is harmless (the frame id is not reset). In case you want to reset the frame id, the third parameter must be a 1, e.g. DB 254, 10, 1.
253	Change of image 	DB 253 DW new_img The image "new_img" is associated, which must be a memory address.
252	Change of route	DB 252, 2, 0 Route 2 is associated
251	Go to next frame from the animation. 	DB 251, 0, 0 The Sprite is animated. The two zeros are fillers

**IMPORTANT:** be very careful to write DB and DW where they should be used, i.e., for example, if you change images you should precede the image with DW and not with DB. If you make such a mistake, your route will not work.

**IMPORTANT:** A route can be at most 255 bytes long and a segment is 3 bytes long, so a route can be at most 84 segments long. You may need to build an even longer route and in that case you can do so by concatenating the end of a route with a route change to another route (code 252), and you can concatenate as many routes as you wish.

**IMPORTANT:** escape codes can be used in the middle of a route, but the last segment cannot be an escape code, it must be a movement, even if it is standing still, something like "DB 1,0,0".

In these cases, |ROUTEALL will interpret that a change of state, sequence, image, sprite path or animate must be forced and also execute the next step. Changes can be forced at any segment of the path, not necessarily at the end, and you can force as many changes as you want.

```
ROUTE0; one shot to the left
;-----
db 100,0,-1; one hundred steps left at speed Vx=-1 db
255,0,0; deactivation of sprite with status=0
db 1,0,0 ; do not move anything in this
step db 0
ROUTE1; one jump to the right
db 253
dw SOLDIER_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 SOLDIER_R1_DOWN
db 1,-5,1; up so that UP and down match db
2,1,1
db 2,2,1
db 2,3,1
db 2,4,1
db 1,5,1
db 253
dw SOLDIER_R1
db 1,5,1; down one more
db 255,13,0; new state, without path flag and with animation flag
db 254,32,0; macro sequence 32
db 1,0,0; quietooo.!!!!
db 0
```

### 21.1.22 |ROUTESP

This command allows you to route a single Sprite that has the route flag active in its status byte through its assigned route (SETUPSP parameter 15).

Use:

**|ROUTESP, <spriteid>, <steps>.**

**|ROUTESP, <spriteid> : rem in this case "steps" is considered=1** This command moves a sprite any number of steps (**up to 255**) along its assigned path. The command moves the sprite through all the steps indicated, leaving it finally located in the same position it would have had if we had executed |AUTOALL,1 a number of times equal to the number of steps.

IMPORTANT: steps cannot take a value greater than 255

### **21.1.23 |SETLIMITS**

This command sets the limits of the area where sprites or stars can be printed.

Use:

**|SETLIMITS, <xmin>, <xmax>, <ymin>, <ymax>, <ymin>, <ymax>, <ymin>, <ymax>, <ymax>, <ymax>.**

Example that sets the whole screen as the allowed area

**|SETLIMITS,0,80,0,200**

Outside these limits, the sprites are clipped, so that if a sprite is partially outside of the area allowed, the functions |PRINTSP y |The PRINTSPALL will print only the part that is within the allowed area.

### **21.1.24 |SETUPSP**

This command loads data from a sprite into the SPRITES\_TABLE

Usage:

**|SETUPSP, <id\_sprite>, <param\_number>, <value>.**

Example:

**|SETUPSP, 3, 7, 2**

Allows for example to assign a new animation sequence when the sprite changes direction, or simply to change its status flags register.

With this function we can change any parameter of a sprite, except X, Y (which is done with LOCATE\_SPRITE).

We can only change one parameter at a time. The parameter to be changed is specified with param\_number. The param\_number is actually the relative position of the parameter in the SPRITES\_TABLE

Param number	Action	Possible use of POKE or  POKE as an alternative
0	changes the status (occupies 1 byte)	YES
5	changes Vy (occupies 1byte, value in vertical lines). You can also change Vx at the same time if we add it at the end as a parameter	YES
6	changes Vx (occupies 1byte, horizontal byte value)	YES
7	change sequence (occupies 1byte, takes values 0..31)	NO, because  SETUPSP also resets the frame_id and also gives you assigns the address of the first image.
8	change frame_id (occupies 1byte, takes values 0..7)	YES

9	change dir image (occupies 2bytes). The specified image can be one of the initial list of images in the images_mygame.asm file,	It is not the same if an image <255 is used. If a memory address is used,  POKE could be used as an alternative.
15	change the path (takes 1bytes)	<b>NO</b> , because SETUPSP does more things in internal tables to make the route work.

Example:

In this example we have given sprite 31 the image of a ship that is assembled at address &a2f8.

```
ship = &a2f8
|SETUPSP, 31, 9, nave
```

There is an easier way to specify the image for the sprite by making use of the IMAGE\_LIST in the images\_yourgame.asm file. If we have the NAVE in the IMAGE\_LIST, we can associate an identifier between 16 and 255

|SETUPSP,31, 9, 16 : rem the 16 is the identifier of the NAVE in the IMAGE\_LIST

#### IMAGE\_LIST

;-----  
We will put here a list of the images we want to use without specifying the memory address from basic.

The command |SETUPSP,<id>,9,<address> becomes |SETUPSP,<id>,9,<number>; thus the command |SETUPSP,<id>,9,<address> becomes |SETUPSP,<id>,9,<number>.

The advantage of not using memory addresses in BASIC is that if we enlarge the graphics or reassemble them in The number we assign will not change.

They do NOT all have to have a number, only those we are going to use with |setupsp, id, 9,<num>.

The numbering starts at 16

We can use up to 255 images specified in this way.

The list does not need to be 255-elements long, it is of variable length, it can even be empty.

;

**DW NAVE ; 16**

**DW OTHER\_SHIP ; 17**

;----- BEGIN IMAGE -----

NAVE

db 7 ; width

```
db 12 ; height, 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 , 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 -----

In the case of param\_number=5, we can include Vx as a parameter at the end:  
|SETUPSP, 31, 5, Vy, Vx

This way we will update the two speeds with one command, which costs 3.73ms as opposed to the 6.8ms it would take to invoke two commands separately.

In the case of using param\_number=7, besides changing the animation sequence, the command automatically updates the frame (frame id), placing it in the initial one (the zero) and the image address is updated, so you don't need to invoke param\_number=9 to change the sprite image to the first image of the new assigned sequence. If you are using |ANIMALL before printing, or

|Even though SETUPSP will place the animation at frame zero, you will jump to frame 1 before printing. This is normally not a problem, but in the case of a death sequence where for example the first frame is to delete the sprite, you may not want to jump directly to frame 1. In that case a simple trick can be to repeat frame zero in the definition of the death sequence. That way you make sure that the frame is visible. Another option is to remove the animation flag and animate it with ANIMASP after printing.

In the case of param\_number=15, in addition to assigning the path to the sprite in the attribute table, the command performs an internal data reset so that the sprite starts traversing that path from the first segment of the path in question.

### 21.1.25 |STARS

Moves a bank of up to 40 stars on the screen (within the limits set by |SETLIMITS), without painting over other sprites that were already printed.

**|STARS,<initial star>,<num stars>,<colour>,<dy>,<dx>,<dx>.**

Example:

**|STARS, 0, 15, 3, 1, 0**

The example shifts 15 stars of colour 3 (red) one pixel vertically (since dy=1 and dx=0). Repeatedly invoked gives the sensation of a scrolling star background. When a star leaves the screen boundary or the one set by |SETLIMITS, it reappears on the opposite side, so there is a sense of continuity in the flow of stars.

The star bank is located at address 42540 (=&A62C) and has a capacity of 40 stars up to address 42619. Each star consumes 2 bytes, one for the Y-coordinate and one for the X-coordinate.

Groups of stars can be moved separately, starting at the star of your choice. The initial coordinates of the stars must be initialised by the programmer.

Example of initialisation and use in a scroll of four star planes to give a sense of depth. Each plane will move at a different speed

**1 MEMORY 24999**

**10 CALL &6b78: rem installs RSX commands**

```

20 bank=42540
30 FOR star=0 TO 39:' loop to create 40 stars
40 POKE bank+star*2,RND*200
50 POKE bank+star*2+1,RND*80
60 NEXT
70 MODE 0
80 REM we will paint and move 4 star planes of 10 stars each.
90 |STARS,0,10,3,0,-1: ' 3 is red. The more distant ones move more
slowly
91 |STARS,10,10,2,0,-2: ' 2 is blue
92 |STARS,20,10,1,0,-3: ' 1 is yellow
93 |STARS,30,10,4,0,-4: ' the 4 is white. The closest ones go more
quickly
95 goto 90

```

The uses of this command can be very diverse.

- Using several banks of stars at the same time with different speed and colour can give a sense of depth.
- If the direction of the stars is diagonal you can make a "rain effect".
- If the colour is black and the background is brown or orange you can give the impression of advancing over sandy territory.
- If the motion is a rolling motion and the colour of the stars is white, you can give the impression of snow. The rolling motion can be achieved with a zigzag in X while keeping the speed in Y, or even using trigonometric functions such as cosine. Obviously if you use cosine in game logic it's going to be very slow, but you can store the pre-calculated cosine value in an array. Example of snow effect:

```

1 MEMORY 23999: MODE 0
30 ' initialisation of 40-star bank
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

```

There is a way to achieve faster execution, and that is to avoid passing parameters. Throughout this book we have seen how passing parameters is expensive even if the command invoked does nothing. Well, we are dealing with a command that requires 5 parameters, so it is especially expensive. If we want to reduce the time required by BASIC to interpret the parameters, we can simply invoke the command once with parameters and the following times without passing parameters.

**|STARS,0,10,1,1,5,0**

**|STARS** : this parameterless invocation assumes the same values as the last invocation.

This possibility is especially useful in games where we want to invoke the command every game cycle to move stars, as it will save about 1.7 ms.

**IMPORTANT:** The STARS command is affected by the **|SETLIMITS** limits but only if Vx or Vy are non-zero. If both are zero then **|SETLIMITS** is not affected and stars can be painted all over the screen.

### **21.1.26 |UMAP**

This command updates the map with information located in another memory area where we have a larger map. The command causes the entire map to be rebuilt, including only those items that meet certain X, Y coordinate ranges (all parameters are 16-bit numbers).

Use:

**|UMAP, <map\_ini>, <map\_fin>, <y\_ini>, <y\_fin>, <x\_ini>, <x\_fin>, <x\_fin>, <x\_fin>, <x\_fin>, <x\_fin>, <x\_fin>, <x\_fin>, <x\_fin>, <x\_fin>.**

For example, if we have a map located at address 22.000 that occupies 1500bytes and we want to update the map with the coordinates of our character, with enough margin to advance in the Y-coordinate up to 100 lines and in the x-coordinate up to 20 bytes in all directions:

**|UMAP, 22000, 23500, y-100, y+100, x-20, x+20**

This command will check the coordinates of the items located on the map at address 23000 and if they are within the X, Y margins that we have set, they will be copied to the memory area that 8BP uses for the |MAP2SP command, that is, it will copy them from address 42040. However, it will only copy those that meet the condition. As there are fewer items, the |MAP2SP command will run faster as it will have to read and check if there are fewer items on the screen.

**IMPORTANT:** the map to be copied must NOT include the 3 parameters of every map:

<Max\_high> ( which is a DW , i.e. 2 bytes)  
<max\_width> ( which is a DW , i.e. 2 bytes)  
<Num\_items> ( which is a DB , i.e. 1 byte)

That is, there are 5 bytes that should not be included in the map to be copied.

## 22 How to make a scoreboard

In many games it is interesting to have a scoreboard mechanism (also called "Hall of fame") that stores the best scores in an orderly fashion from different games. It can be programmed in BASIC by means of an array that stores the score of each game, but every time we stop the game and RUN, the BASIC interpreter internally executes a CLEAR and all the values of this table are erased. One way to avoid this is to prevent the user from interrupting the program by pressing the ESC key twice using the firmware routine CALL &bb48. Another option is to store the points in a table in memory and read and modify it from BASIC. This can be programmed in many ways, and here is an example. The steps to be taken are:

Include in make\_all.asm a reading of the file "score\_table.asm".

```
;-----CODIGO-----
;includes the 8bp library and the music playerWYZ
read "make_codigo_mygame.asm".
;-----MUSICA-----
read "make_musica_mygame.asm";
includes the songs.
; ----- GRAPHICS -----
; this part includes images and animation sequences read
"make_graficos_mygame.asm".
read "score_table.asm"
```

Next, we need to create such a score\_table.asm file with sample data. I have created one with Sumerian goddess names and scores from 10 to 1. Each name takes 8 characters. Something very important is the **org\_end\_graph**. With this command we are indicating that the table is going to be assembled after the graphics, in the memory addresses that follow.

```
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
```

Now comes the BASIC part: We will assemble with winape and then we will look (with winape, option assemble->symbols) in which memory address the end\_graph tag has been assembled. In my case it has been &9685. In our BASIC program we will we will take into account ( I store it in the 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

```

Let's look at the routine that reads the score table in line 2040.

```

2040 '--- READ SCORE TABLE
2050 dir=&9685: FOR i=0 TO 9: name$(i)=""
2070 FOR j=dir TO dir +7:'lee character a character the 8 letters
2080 letter=PEEK (j):
name$(i)=name$(i)+CHR$(letter)
2090 NEXT j: dir=dir+8:'after the 8 letters there points (an integer)
are the
2100 pts(i)=0: | PEEK,dir,@pts(i):dir=dir+2:'an integer is 2 bytes
2110 NEXT i
2120 RETURN

```

Finally, every time a game is over, we check if the score (variable "score" in the example) is higher than any record in the score table (array "pts") and if so, we modify the table. By modifying the table in the memory addresses, we will not lose the values, even if we RUN.

```

1800 '--- END GAME & 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 moves all lower scores one position.
1840 FOR i=8 TO j STEP -1: pts(i+1)=pts(i): name$(i+1)=name$(i): NEXT
1850 b$=INKEY$:IF b$<>"" THEN 1850:'clean buffer keyboard
1860 MODE 1: BORDER 5: INK 3,8: LOCATE 6,8: PEN 3: PRINT "CONGRATULATIONS!
NEW HIGH SCORE".
1880 LOCATE 14,10: PEN 2: PRINT "ENTER YOUR NAME".
1900 LOCATE 15,12: PEN 1: INPUT name$(j): name$(j)=MID$(name$(j),1,8)
1910 pts(j)=score
1920 '--- WRITE SCORE TABLE ON MEMORY --
1930 dir=&9685: FOR i=0 TO 9: k=1
1950 FOR j=dir TO dir +7:'we write character by character, all 8 letters
1960 dato$=MID$(name$(i),k,1): IF dato$="" THEN dato$=" "
1970 dato=ASC(dato$)
1980 POKE j,data:k=k+1:NEXT j
1990 dir=dir+8: 'we write the punctuation after the name (8 letters)
2000 |POKE,dir,pts(i)
2010 dir=dir+2:'the score is an integer = 2 bytes
2020 NEXT i
2030 RUN

```

## **23 Possible future enhancements to the library**

The 8BP library can be improved by adding new functions that could open up new possibilities for the programmer. Here are some suggestions for doing so

### **23.1 Memory for locating new functions**

Currently, through the mechanism of the "assembly options", the library leaves you an amount of free memory for your BASIC listing that depends on each option.

- Option 0: 23.5 KB free (should only be used to test things)
- Option 1: 25 KB free (maze games)
- Option 2: 24.8 KB free (games with scroll)
- Option 3: 24 KB free (games with pseudo 3D)

The library could still grow, through an option 4 that extends the capabilities of option 1, e.g. through "filmation" capabilities. This option 4 could provide such capabilities using 1 KB and leaving the user 24 KB free.

### **23.2 Pixel resolution printing**

Currently 8BP uses byte resolution and byte coordinates, which are 2 pixels of mode 0. Actually, one way around this limitation is by defining 2 images for the same sprite that are offset by a single pixel. When moving the sprite around the screen you can alternate between simply changing the image to the offset image and moving the sprite by one byte. That way you will get a pixel by pixel movement. This technique is detailed in chapter 13

### **23.3 Layout de mode 1**

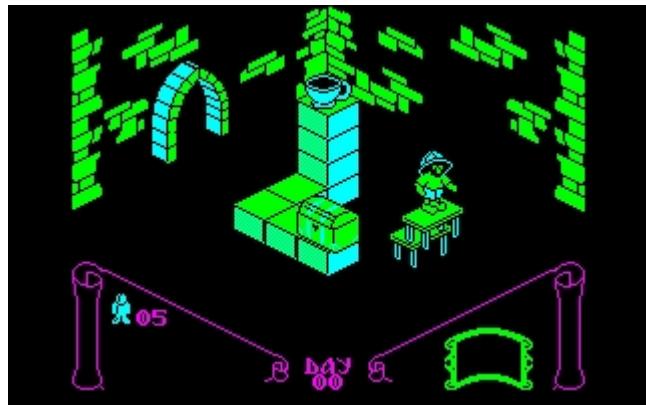
The current layout works as a character buffer of  $20 \times 25 = 500$  Bytes. It can be used in mode 1 games without problems, but there will be things we can't do, like defining a part that takes up 3 characters of mode 1 width, as mode 0 characters take up twice the width of mode 1 characters. It's not a problem, but it is a limitation.

A mode 1 layout would occupy 1KB so  $40 \times 25 = 1000$ . Since the mode 0 layout and the mode 1 layout would not be used simultaneously, they could overlap in memory and taking into account that the mode 0 layout is in 42000 to 42500, we would simply place the mode 1 layout between 41500 and 42500, "stealing" 500bytes from the 8KB sprite memory, located between 34000 and 42000.

The changes to support this enhancement are minimal, affecting only two features. The layout0/layout1 and **|LAYOUT** and **|COLAY** should be aware of the screen mode, by means of a variable that acts as a flag (layout0/layout1). This modification might be fine but even without it, we can still use layouts in mode 1 games without problems.

### **23.4 Filmation capacity**

It could be interesting to create a "filmation" mode for making "knight lore" type games. By making use of existing functions in the library, with a little extra code this interesting capability could be implemented. It is possible that this capability will be added soon.



*Fig. 120 the mythical "Knight Lore".*

### 23.5 Hardware scroll functions

There are few Amstrad CPC games with quality smooth scrolling, programmed using the capabilities of the M6845 video controller chip. Currently the library has a CPU based scrolling mechanism (not hardware based but efficient and versatile for games with scrolling in any direction of movement).

The fact that there are not many games like this is due to the fact that in the 1980s, videogame programmers did not have much information and in many cases they were amateurs.

Among the few games that have smooth scrolling are 2 from Firebird:

- "Mission Genocide" (from Firebird, 1987, by Paul Shirley, an excellent programmer who also invented an ultra-fast overwriting technique without the use of masks).
- "Warhawk" (by Firebird, 1987)



*Fig. 121 Firebird games with fast and smooth scrolling*

The scrolling technique of these two games is the same, known as "vertical scrolling". The scrolling technique consists of controlling exactly the instant at which the screen scroll occurs. At that moment we trick the CTRC 6845 by telling it that the screen ends earlier than normal. However, before ending that section of the screen, we tell it to incorporate fewer scanlines than correspond to a section of that size. Then, at a very precise instant that we must control to the microsecond, we tell the chip to start a new screen, without having produced the vertical sync signal. This allows us to draw a second

screen (the markers, for example) and compensate the number of scanlines of the first section. If we get the scanline compensation mechanism right, we can make one of the two screen sections move extraordinarily smoothly. The problem with taking this technique to a command to be used from BASIC is that the control of the interrupts is imprecise due to the execution of the interpreter, and here we need very, very precise control.

The problem with hardware scrolling (which also affects software scrolling that moves the whole screen) is that it "drags" the sprites present with it, so that when you reposition them you will notice an unwanted vibration in the enemies and/or in your character. To solve this, you can use double buffering and switch between two 16KB blocks each time a frame is ready. This will prevent you from being able to see "how each frame is done". In 8BP I discarded double buffering in order to leave a good RAM space for the programmer and that's why these techniques have not been implemented.

For all the above reasons, scrolling in 8BP is based on a world map that does not drag the sprites when moving and is therefore more efficient as it moves less memory while allowing multidirectional movement.

### **23.6 Migrating the 8BP library to other microcomputers**

This library would be easily portable to other Z80-based microcomputers, such as the Sinclair ZX Spectrum. In the case of the ZX spectrum it would be necessary to rewrite the routines that paint on the screen, as the video memory is handled differently. The ZX migration is already a firm project, after having received numerous requests from ZX users.

The migration of the library to a Commodore 64 would also be feasible, although the assembly code could not be reused, since it is based on another microprocessor. Furthermore, in the case of the commodore 64, the migration of the 8BP library should take advantage of the machine's own features such as its 8 hardware sprites, so that what the 8BP library should incorporate internally would be a sprite multiplexer, offering 32 sprites, but internally using the 8 hardware sprites.



*Fig. 122 Sinclair ZX and Commodore 64, two classics*



## 24 Some games made with 8BP

In this chapter I am going to describe how some games that you can find on the web <https://github.com/jiaranda13/8BP> made with 8BP (from the most recent to the oldest) are made:

- **Paco the man:** a puzzle-style game, which makes use of the soft move technique (half byte) and advanced massive logics.
- **NOMWARS:** a "commando" style game
- **Blaster pilot:** a multi-directional scrolling game and is inspired by the style of games like "Time Pilot" or "Asteroids".
- **Happy Monty:** fast-paced mutant monty style game
- **Eridu:** a classic scramble-style game with horizontal scroll.
- **Space phantom:** inspired by space harrier
- **Eternal Frogger:** a remake of the classic frogger, presented at the famous "eternal amstrad" fair.
- **3D Racing one:** first racing game to use pseudo 3D capability
- **Fresh Fruits & vegetables:** a platform game using horizontal scrolling and advanced sprite path management.
- **Nibiru:** a horizontal scrolling ship game, using advanced features of 8BP
- **Anunnaki:** a ship game, arcade genre
- **Mutante Montoya:** a screen-scrolling game. It could be classified as a platformer. It was the first game I made with 8BP.
- **Mini-games:** these are short, simple, didactic games to get you started in programming with 8BP. There is a version of the classic "pong", called "Mini-pong" and a version of the classic "Space Invaders", called "mini-invaders".

### 24.1 Mutant Montoya

A first tribute to the Amstrad CPC, with a title inspired by the classic "mutant monty".

It is a simple 5-level game. It is based on the use of the 8BP layout to build each screen.





## 24.2 Anunnaki, our alien past

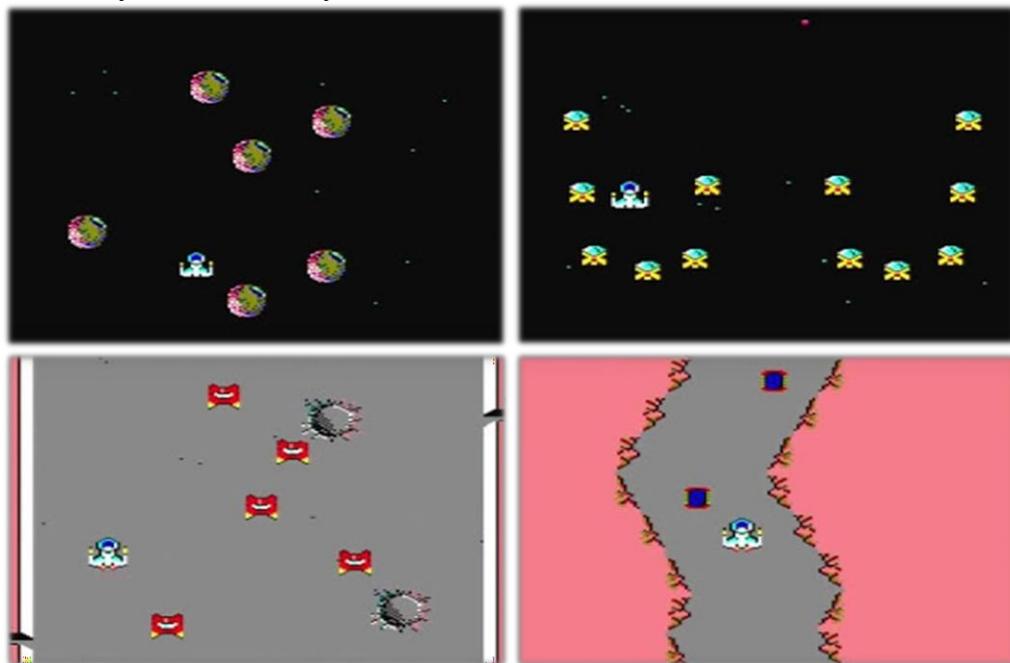
This is a very interesting arcade videogame to analyse and learn about the "massive logic" programming technique. When it was programmed, the 8BP library did not yet have a scroll command or sprite routing, which is why the programming of this game is so interesting, as it achieves everything through massive logics.

Unlike "Mutant Montoya", the video game "Anunnaki" does not make use of the layout, as it is a game about advancing and destroying enemy ships, it is not a game of mazes or passing screens. This game also makes use of "simulated" scrolling, very interesting.

You are Enki, an Anunnaki commander who confronts alien races to conquer planet Earth and thus subjugate humans to their will.

The game consists of 2 levels, although if you lose a life, you continue at the point in the level where you are, you do not go back to the beginning of the level.

The first level is a stage in interstellar space, where you must dodge meteors and kill hordes of ships and space birds. At the end of the level you must destroy a "boss". The second level takes place on the moon, where you must destroy hordes of ships, after which you must go through a tunnel riddled with mines until you encounter three "bosses" that you must destroy.



## 24.3 Nibiru

This is a game that tests many of the features of 8BP and the "massive logic" programming technique, and has details like a fancy load graph and three in-game tunes, as well as a score table that doesn't get lost even if you restart the game and other advanced technical aspects like parallax scroll, routes, macros, etc. The BASIC listing is just over 16KB.

You are the pilot of a destroyer ship and must defeat the planet Nibiru and its leader, "Gorgo", an almost invincible ancient reptile. You must destroy the galactic birds that live on its moons and once you reach the planet you must face its dangers before you can fight Gorgo.

The game consists of three phases and uses 8BP's scrolling mechanism based on the MAP2SP command and also uses sprite routing and animation macros - all from BASIC! thanks to 8BP and the "massive logic" technique.



The game maintains a score table that is not deleted, even if you stop the game. This is achieved by storing it in RAM with pokes, instead of storing it in BASIC variables.

## 24.4 Fresh Fruits & vegetables

This is a platform game, where your mission is to collect all the fruits to leave the population with nothing to eat, so that they have to sacrifice a poor pig to feed themselves.

Its main novelty is the advanced use of routes, concatenating one to another to go from "falling" to "walking" and the use of RINK combined with MAP2SP as a scrolling technique.



The blue bricks in the game presentation use synchronised sprite printing (PRINTSPALL,0,0,1) while during the game the sprite printing is not synchronised (PRINTSPALL,0,0,0). The effect of synchronisation on the presentation is that everything appears synchronised, including the RINK ink animation (used on the blue bricks). This gives a smooth scrolling effect. In my opinion, you should not sync a game because, although you get more smoothness, the speed of the game also decreases. Depending on the type of game you play, you may or may not be interested.

## 24.5 "3D Racing one

This is the first game made using 8BP's pseudo3D capability. It has a fancy loading graphic and 4 tracks: a training track with puddles on the road, a track where we compete with 4 other cars, a track where the speed is doubled, using lower slope segments, and a final night stage. The game also uses the PRINTAT command and its own character set. In addition, it has a scoreboard, a main music track, two secondary tracks and sound effects.

For the presentation a 2D map full of balls is used and the MAP2SP command with overwrite is configured. The balls are "zoom images".

The first circuit has been built with a file in which we have placed one by one all the elements (segments, signs, trees, puddles...) but the rest of the circuits are created dynamically from BASIC, poke the address of the world map.

To detect collisions and road departure, the PEEK command is used instead of COLSPALL, so that as soon as it detects a byte on the car of a colour other than the road, you are considered to be in collision and your engine is damaged.

Another interesting new feature is the use of dynamic routes. Both the competition circuit and the car routes are created from BASIC, following the procedure explained in this manual.



## 24.6 Space Phantom

This is a game made with the v35 version of the library. It uses the pseudo-3D capabilities for the "Star Wars" style presentation of titles. It also uses transparent printing with sprites (coins) passing behind the background (the scoreboard).

The game is inspired by the classic "Space Harrier" and you control a hero equipped with a jet-pack who flies through space, killing meteors, UFOs, space birds and a dragon as the final enemy. It consists of three stages and an epic finale.

The character set is in-house, although only the numbers have been defined, in the style of a "casio clock" for the markers.

In the first phase, routes are used for the stars, which are sprites as well as the meteors and birds.

The second uses sprite overwrite with 2-bit background (4-colour) and ink animation using RINK. The ships in a row are built by placing them using the improved ROUTESP command, available in V35.

Although the game simulates 3 dimensions, it does not use pseudo-3D projection, but rather sprite paths in which the enemy version is changed to a larger version to give the impression that it is approaching. So that a collision with a distant enemy does not affect us,

an unused flag status register is used to mark distant enemies as harmless. In the third phase, relative horizontal movement of the stones on the ground has been used in combination with an accelerated vertical movement path.



## 24.7 Eternal Frogger

The game "Frogger Eternal" has been made with the V36 version of 8BP, and its title evokes both the classic "frogger" by konami released in 1981 and the "Amstrad Eternal" fair held in 2019, the event where this game made its appearance.

It is a game programmed in mode 1, using LAYOUT, transparent printing on the frog and routes of various nature for the sprites. Some of the sprites are invisible, but collisible, such as 4 "invisible" rivers under logs, water lilies and turtles that the frog must jump over, like "invisible walls" on the sides of the river, so that the frog cannot escape.



## 24.8 Eridu: The space port

This game is a clone of Konami's classic "Scramble" created in 1981. It actually has many differences from the original, but in essence it is inspired by the classic game, as it incorporates the need to refuel, forcing the player to take risks to destroy the fuel tanks so as not to lose a life.

It is quite fast despite running with a powerful scroller, displaying 32 sprites on screen at many points. It reaches up to 18 fps

The game has 5 stages, and different on-game music, as well as a very fancy presentation graphic.

Eridu is a video game that connects to an ancient "forbidden" history of mankind. Eridu was the world's first city, created by the "Anunnaki" 400,000 years ago, an extraterrestrial race according to Sumerian tablets found in the desert of Iraq. There they established a spaceport called "Earth 1".



The maps of the different phases are loaded into different memory banks, each occupying 500 bytes of world data and 200 bytes for the description of enemy locations. The scrolling is logically done with the **|MAP2SP** command.

## 24.9 Happy Monty

It's a fast-paced screen-swiping game, which almost perfectly imitates the classic Mutant Monty. It even "clones" its initial screen. This game is made with version 37 of the library and reaches 25 FPS.

It makes intensive use of layout and, of course, of massive logics. It manages to store 25 levels using a simple technique to compact the layouts (each layout spends only 160 bytes) explained in this book.



An original technique used in this game allows enemies to change their path. These are invisible "reversing" sprites. When an enemy collides with an inverter sprite, it changes its path and is assigned the opposite path or even a perpendicular path, thus being able to build paths of any length without defining more than one vertical and one horizontal path. Even loops can be made.

This also simplifies the creation of the map (layout + enemies) of each level, because when locating an enemy it is enough to use a code (a character) that indicates the speed and direction of the enemy (6 letters are used for all types of enemies and periodically changes the "palette" of enemies, the dolls associated with those 6 letters).

## 24.10 Blaster Pilot

It is a game created with the v39 version of 8BP, in which it is possible to have sound effects (created with SOUND) at the same time that music is played using the MUSIC command. In this case the SOUND command is used for gunshots and explosions and uses the third channel while the music uses the first two channels.



The game has multi-directional scrolling and is inspired by the style of games like "Time Pilot" or "Asteroids". Next to the score and lives panel there is a radar with which you can orientate yourself in space to locate 3 lost astronauts that you have to rescue. The game is interesting in how the programming of the ship's movement in 12 possible directions is dealt with as efficiently as possible.

In addition to 6 levels, it has a bonus phase at the end of each level, which allows you to accumulate points and recover a life.

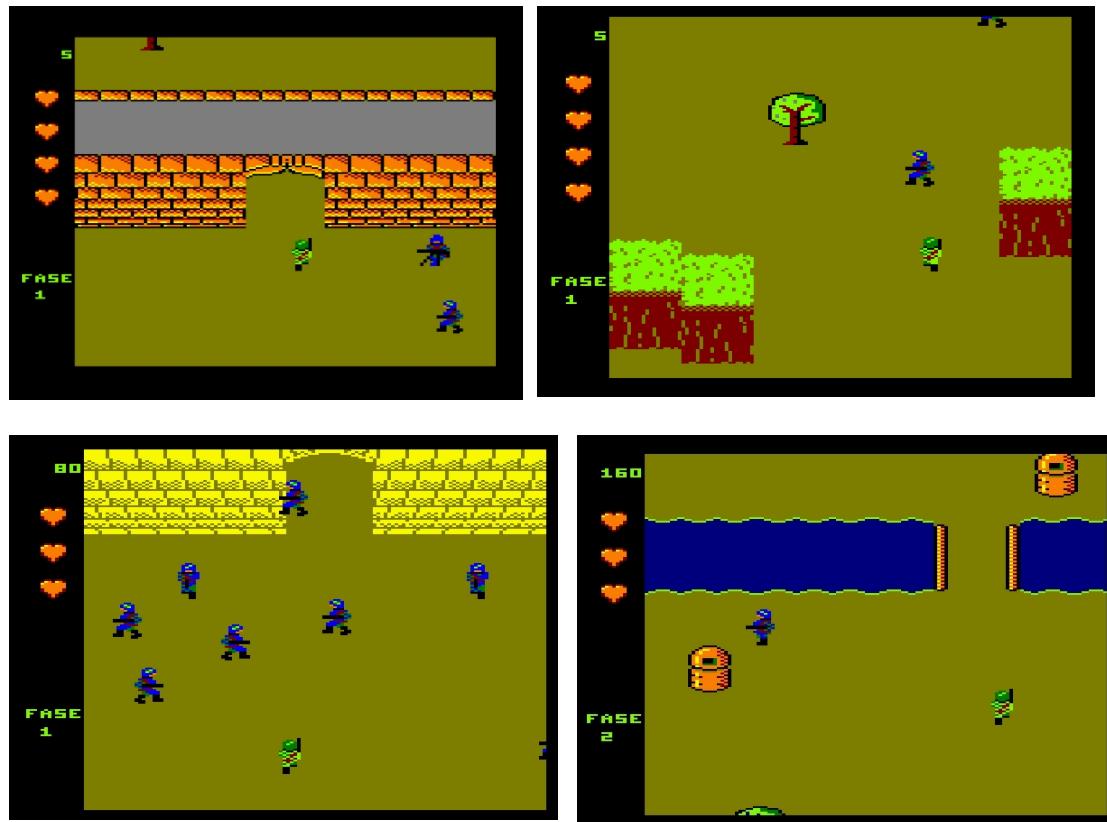
## 24.11 NOMWARS

It is a game in the style of the classic "Commando" created in 1985 by Capcom. A classic vertical scroll shoot 'em up.

The game consists of 4 stages and a "fancy" intro, with a story about the new world order war. It has been published in DES format.

This version exploits the scrolling capability of 8BP (MAP2SP command) and includes the famous bridge that Joe passes under using a technique based on 8BP's SETLIMITS command.

The game is offered in two versions: the pure BASIC version and the compiled cycle version (cycle translated into C language using the 8BP wrapper and the 8BP minibasic). Both versions are identical, as the C translation is a total replica, almost a mirror of the BASIC version. In fact, the game has been programmed in BASIC and the last day it has been translated to C with the facility that 8BP has for it.

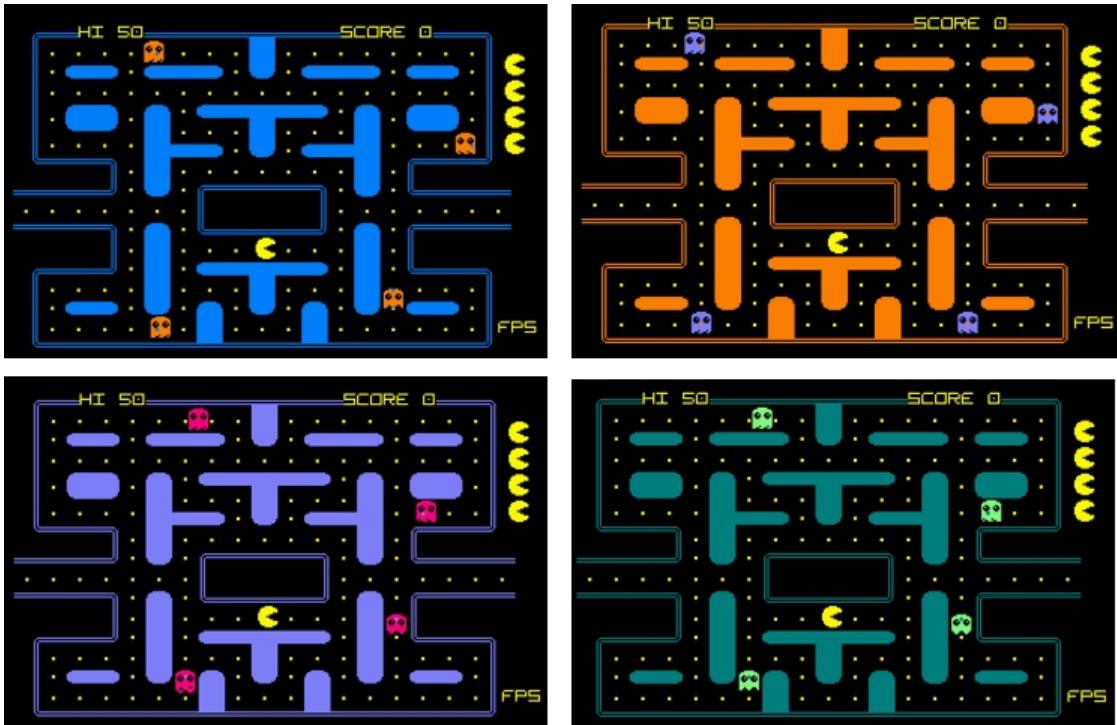


## 24.12 Paco, the man

A game in the purest Pac-Man style. The game contains two phases in BASIC and two with compiled cycle. In BASIC it reaches 19 FPS with maze, collisions and 4 ghost logic and in C it reaches 33 fps.



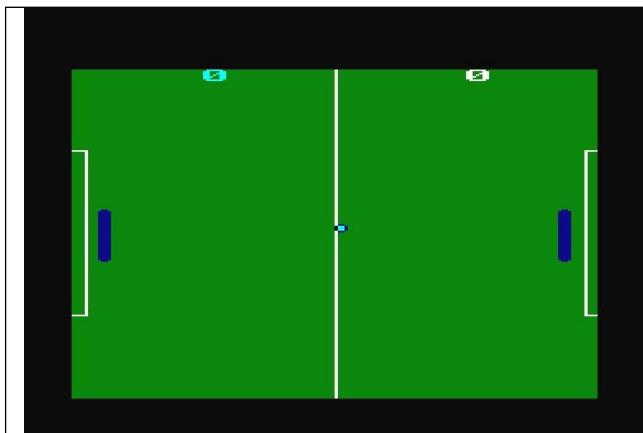
Each Paco level is identified by its colours. The first two run in BASIC and use "pre-calculated" ghost decisions to reach 19 fps. The third and fourth stages use C and reach 33 fps.



### 24.13 Mini Games

These are games made for educational purposes. Simple to understand and short, to help the novice programmer in the development of their own games.

#### 24.13.1 Mini-pong



*Fig. 123 "Mini-pong" video game*

It is a very simple and didactic videogame. Based on the classic "Pong" by Atari (1972). The opponent's bar (the computer) starts to make decisions when the ball goes over half the field, so it is possible to beat him. If we put him to make decisions earlier, there comes a time when it is impossible to win.

A few brief details about the game:

- Use of **|COLSPALL**: to detect collisions between the "collider" (the ball) and the "colliders" (the bars). In the status byte of the sprites, the collider flag is activated for the ball (sprite 29) and the collider flag is activated on 31 (our bar) and 30 (the opponent's bar).
- Use overwrite on the ball to respect the white stripe on the field and not erase it when passing. To do this, use a palette with overwrite and activate the overwrite flag in the status byte of the ball sprite (the 29th).
- there are only two images (the ball=17 and the bar=16) that are assigned to the 2 sprites (sprites 30 and 31 are assigned image 16 and sprite 29 is assigned image 17).
- The sprites use automatic movement. For this they have the automatic movement flag activated and the **|AUTOALL** command moves them (changes their coordinates) according to their speed.
- All sprites are printed with **|PRINTSPALL** in each game cycle.

#### 24.13.2 Mini-Invaders

Like "Mini-pong", this is a game made for educational purposes, inspired by the classic "Space Invaders" by Taito (1978).



Fig. 124 Video game "mini-invaders".

A few hints on how it is done:

- The game uses 32 sprites
- The ship is sprite 31
- The shots you can fire with the ship are 29 and 30.
- Invaders shoot using sprite 28
- Invaders use sprites 0 to 27 (28 invaders in total).
- Sprites 31, 30 and 29 have active collider flag.
- The rest of the sprites are "collided" and have active collision flag.
- Invaders have an active automatic movement flag and are associated with the route "0" which moves them from right to left and down, typical of invaders.
- The ship and invader triggers use a feature of the V27, they traverse the screen and on exit are automatically deactivated with a defined state change at the end of their path, thus simplifying the BASIC logic and therefore speeding up the game.

# 25 APPENDIX I: Video Memory Organisation

## 25.1 The human eye and the resolution of the CPC

The video memory of the Amstrad CPC has 3 modes of operation. The most used mode for games is mode 0 (160x200) because it has more colour, but mode 1 (320x200) has also been used a lot for programming games. Mode 2 (640x200) was rarely or never used for games due to its limited 2 colours.

Since the amount of video memory is the same, resolution is sacrificed in order to gain in the amount of colours, but curiously, horizontally, which is the longer side of the screen, there is less resolution than vertically (160 horizontally and 200 vertically). You may wonder why. Besides, it is not the only microcomputer that did that, many other computers also used the same strategy with the horizontal side.

The reason has to do with the functioning of the human visual system. The eye perceives more detail vertically than horizontally, so "damaging" the resolution on the horizontal axis is not as serious as damaging it vertically. Subjectively the result is more acceptable. The human visual system is like mode 0, very wide pixels. That's why the horizontal field of view is larger than the vertical one.

## 25.2 Video memory

The most complete and clear information can be found in the Amstrad firmware manual. This information will be useful if you want to build an improved sprite editor or want to get into assembler and program print overwrite routines or anything else.

### 25.2.1 Mode 2

In mode 2, each pixel is represented by one bit. So one byte represents 8 pixels. If we take any byte from the video memory, its correspondence with the pixels is 1 bit for each pixel, in this table the bits are represented and to which pixels (p) they correspond.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0	p1	p2	p3	p4	p5	p6	p7

Bit 7 is numbered as the leftmost bit in a byte. Pixel 0 is also the leftmost pixel, i.e. there is nothing "upside down" here. Everything is correct

### 25.2.2 Mode 1

In mode 1 we have 4 colours (represented by 2 bits). One byte therefore represents 4 pixels. The correspondence between pixels and bits is somewhat more complex. Pixel 0 for example is encoded with bits 7 and 3.

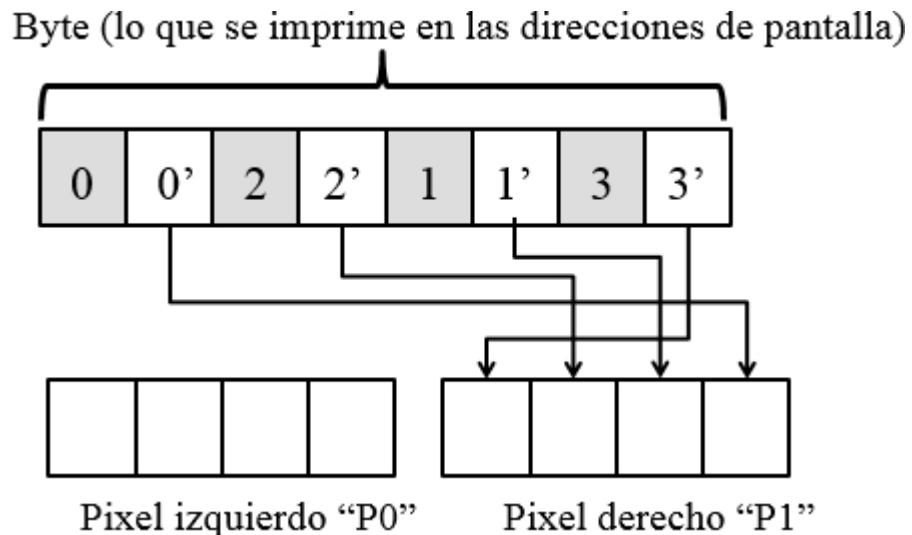
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(0)	p1(0)	p2(0)	p3(0)	p0(1)	p1(1)	p2(1)	p3(1)

### 25.2.3 Mode 0

Here we have a bit of a mess. Each byte represents only two pixels, of which the correspondence with the bits of the byte is as follows: Pixel 0 is encoded with bits 7,5,3 and 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)

The following image will make it clearer for you:



*Fig. 125 pixels and bits in mode 0*

I can't tell you what the obscure reason is for having organised the memory like this, but I imagine the cause lies in the GATE ARRAY, the chip that translates these bits into a video signal. I imagine that the designer reduced circuitry with this twisted design.

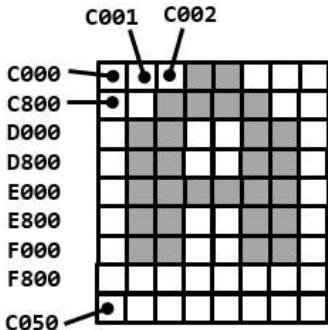
### 25.2.4 Display memory

The screen pixels belonging to the same line are encoded in bytes that are also contiguous. However, from one line to another there are jumps.

If we move forward in memory addresses, when we reach the end of a line we jump to a line that is 8 lines further down. And if we want to continue on the next line, we have to jump in memory addresses to 2048 positions.

The following table shows the video memory. On the left is the line of characters (from 1 to 25) and for each line, the starting address of each of the 8 scanlines that compose it (named as 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	CDF0	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 Screen memory map**

The Amstrad screen is 200 lines x 80 bytes wide each, so the displayed screen memory is  $200 \times 80 = 16,000$  bytes. However, the video memory is 16384 bytes. There are 384 bytes "hidden" in 8 segments of 48 bytes each, which are not shown on the screen, even though they are part of the video memory. These 8 segments are called "spares" in the table above. Each segment is 48 bytes long because, as I said before, to jump from one line to the next you have to add 2048 bytes, but in reality the 25 contiguous memory lines that separate them only occupy  $25 \times 80\text{bytes} = 2000$  bytes.

**From &C7D0 to C7FF both inclusive From &CFD0 to CFFF both inclusive From &D7D0 to D7FF both inclusive From &DFD0 to DFFF both inclusive From &E7D0 to E7FF both inclusive From &EFFD0 to EFFF both inclusive From &F7D0 to F7FF both inclusive From &FFD0 to FFFF both inclusive From &FFD0 to FFFF both inclusive**

You can check this by POKEing on those memory addresses, and you will see that you will not alter the content of the screen.

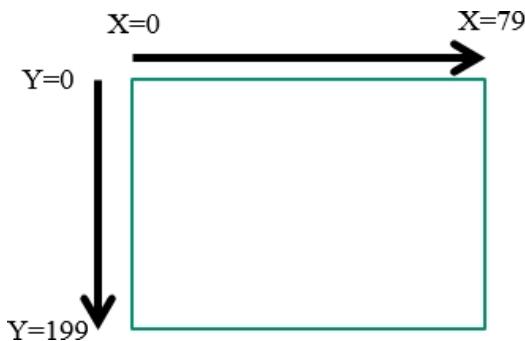
It is tempting to think of using these "hidden" areas of memory to store small assembly routines or variables. However, it is dangerous because a

MODE command executed from BASIC completely erases these memory segments, so if you use it you should be aware of this. In the 8BP library these segments are used to store local variables of some functions, whose value can be safely erased.

### 25.3 Calculation of a screen address

If you want to know the memory address to which specific 8BP coordinates correspond, you will have to perform the following operation

$$\text{Dir} = \&\text{C000} + \text{INT}(y/8)*80 + (y \bmod 8)*2048 + x$$



*Fig. 127 Screen coordinates in 8BP*

This is very useful if for example you want to use PEEK to find out if there is a certain element or colour in a particular byte on the screen and use it as a collision detection mechanism. This technique is used in the video game "**3D-Racing One**".

In case you want to know the direction of some graphical coordinates (the ones used by the BASIC PLOT command) you must first do  $y2=(200-y)/2$ ,  $x2=x/8$

$$\text{Dir} = \&\text{C000} + \text{INT}(y2/8)*80 + (y2 \bmod 8)*2048 + x2$$

### 25.4 Screen sweeps

The Amstrad generates 50 frames per second. That means that approximately every 20ms a new screen scan must be produced.

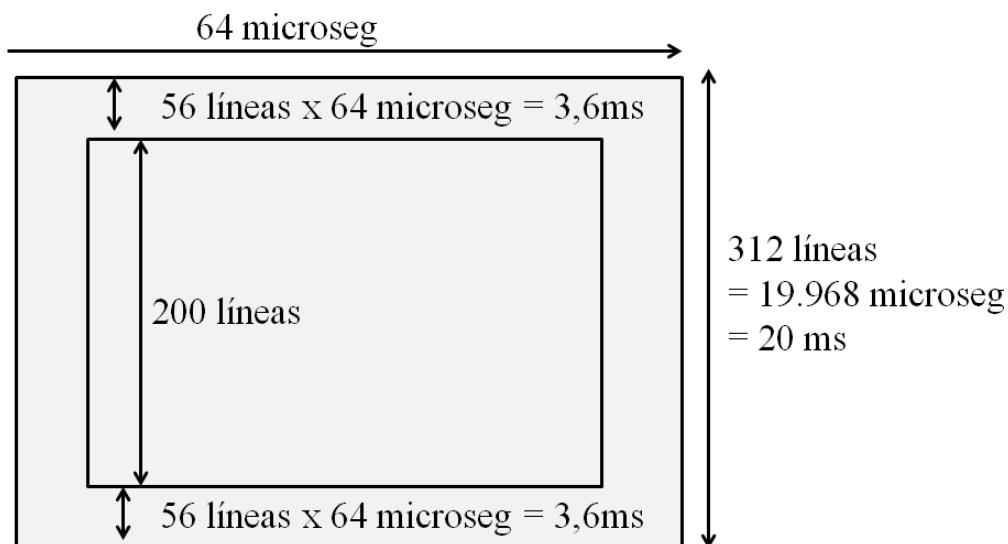
You might think that maybe the screen-swipe, which is painting the screen, consumes a fraction of those 20ms, but it doesn't. Painting the screen takes the Amstrad the full 20ms, so even if you synchronise your sprite printout with the screen sweep it is very likely to catch you, giving rise to two well-known effects:

- **Flickering** occurs when you delete the sprite before printing it in its new position. To avoid this there is a very simple solution: don't erase it. Simply make the sprite erase its own trace, leaving a border on the sprite to fulfil this function. The sprite is bigger but it won't flicker, even if it gets caught in the middle, because it doesn't disappear.
- **Tearing**: occurs when we are caught by the sweep in the middle of the sprite. Half is printed with the new position (head and trunk) and the other half is not.

gives time (the legs). Then the sprite is printed "wrong", although it is corrected in the next frame, but for a moment it is as if it is deformed or broken. Tearing is a bad effect, but much more acceptable than flickering. The perfect solution is to control every millisecond where the flickering is in order to print each sprite without it catching up with us.

A typical recommendation is to print sprites from bottom to top, to minimise these effects. That way it's only possible to get the scan once on one of the sprites, while printing from top to bottom, you can get the scan on several sprites because both (CPU and cathode ray) work in the same direction. Unfortunately the most interesting thing to do is to sort from top to bottom to give depth effect on the sprites (useful in certain games like "Golden axe", "Double dragon", "Renegade", etc.).

The times consumed by the screen are as follows. Note that from the time of the sweep interruption, you have 3.5ms to paint without it being possible to be caught. But in that time you can print at most 2 small sprites.



*Fig. 128 times in a screen sweep*

## **25.5 How to make a loading screen for your game**

There are many ways to do it. A very simple one is to build a graphic with the SPEDIT program modified to allow you to paint all over the screen without showing menus and at the end press some key to launch a SAVE command like this one

**SAVE "mipantalla.bin", b, &C000, 16384**

As you can see the command saves 16KB from the starting address of the screen, which is **&C000**.

The way to load it would be

**LOAD " mipantalla.bin", &C000**

If you do not use the default palette, then you must first execute the INK commands corresponding to the palette you have used, before loading the screen. When loading the screen, you will see how the screen is slowly being drawn on the screen as it loads, since that is precisely where you are loading it, in the video memory.

Another way is to generate a well worked layout and save it using the SAVE command above. The point is to make a drawing and as you can see there are many ways.

Finally, you can use a tool like ConvImgCPC (there are also others), an image converter/editor that works under windows. This tool allows you to transform any image (which can be a scan of a drawing of yours) into a binary file (with .scr extension) suitable for the CPC. This tool also allows you to edit it pixel by pixel and retouch it until you get it perfect. You can even edit it from scratch, without scanning anything. In my opinion this is the best option.

To put this file on a disk (in a .dsk file) you must use CPCDiskXP which is another tool that allows you to put files inside .dsk files.

Once inside the .dsk you can load it with LOAD "mipantalla.bin",&C000  
However, the colours will not look right because ConvImgCPC adjusts the palette to be as close as possible to the original colours. To see them properly you must run the routine where ConvImg places the palette change routine, which is CALL &C7D0.  
This routine is "hidden" in the first of the 8 hidden segments of the video memory, so the .scr image does not occupy more because it contains this routine. The bad thing is that until you don't load the screen you can't execute it and therefore you will see how it loads the image with wrong colours and at the end you will be able to invoke that call, changing the colours. What you can do is to prepare a special palette file. Doing this:

```
Load "image.scr", &c000
Save "palette.bin", b, &c7d0, 48, &c7d0
```

Now you have a 48-byte file containing the palette. In your game loader you would do this:

```
Load "!palette.bin"
Call &c7d0
Load " !image.scr", &c000
```

I recommend that you simply place a few INK commands before the LOAD command that loads the image.

```
Load "image.scr", &c000
```

And just after that, before using 8BP to print sprites, etc. it is convenient to delete the hidden segment where Convimg leaves the routine, because it is a space that 8BP uses for variables and if they are not initially set to zero, they can interfere

```
for i = &c000+2000 to &c000+2000+48: poke i,0:NEXT
```

in a nutshell:

```
10 <several INK commands>
20 Load " !image.scr", &c000
30 for i = &c000+2000 to &c000+2000+48: poke i,0:NEXT
```

To leave the palette at its default values, use CALL &BC02, which is a firmware routine.

**And to save the screen on a tape?**

We've seen how to do it on disk but CPCDiskXP won't leave the .scr file on the tape, so we have to do something like this:

```
|DISC  
MEMORY 15999  
LOAD "image.scr", 16000  
|TAPE  
SPEED WRITE 1  
SAVE "imagen.scr",b,16000,16384
```

And when we load it, we load it in the same way as on disk:

```
Load " !image.scr", &c000
```



## 26 APPENDIX II: The Palette

The following table shows the AMSTRAD palette. Within each colour and in brackets is the ink number assigned to that colour in the default palette. The 27 colours are:

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 (19)	20 - Cyan claro (2)	21 - Verde lima	22 - Verde pálido (13)	23 - Cyan pálido	24 - Amarillo claro (1)	25 - Amarillo pálido	26 - Blanco (4)

The default palette values in each mode are:

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

The palette values in each mode are managed with the INK command, see the Amstrad BASIC reference manual for more information. For example, to set the zero ink as red, we consult the palette of the 27, see that red is the sixth colour and write

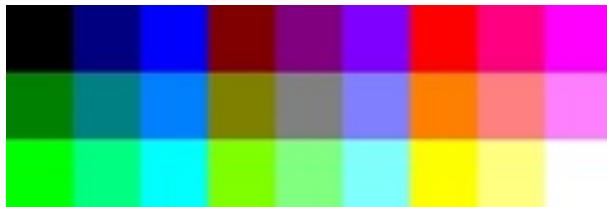
INK 0,6

And we have already configured the zero ink to be red. As you can see, an ink is not a specific colour, but can be configured to be any colour you want.

The reason why the Amstrad palette is so good is because the 27 colours offer so many possibilities, even though only 16 can be used at a time. The colours are sorted by brightness.

If we represent the colours of the Amstrad in 24bit RGB scale (8 bits per component), we find that it has 3 values of red, 3 of green and 3 of blue, (these values are 0,127 and 255) being the number of combinations  $3 \times 3 \times 3 = 27$ . The grey colour is right in the centre of the palette, where the values of R,G,B are equal.

(R=127,G=127,B=127). The 3 components are also the same in white (R=255,G=255,B=255) and black (R=0,G=0,B=0).



*Fig. 129 Amstrad palette*

Having a palette of 27 means that, although we can only choose 16, there are always colours to choose from, allowing us to create fades and blends. However, other computers of the time, such as the C64 (a great machine) had 16 colours out of a palette of 16. The C64 had 3 shades of grey within such a reduced palette, which, although it has been criticised, I think is not bad because as they are not very saturated colours they combine well with the grey, and they also combine well with each other, that is, they are 16 colours that are "close" to each other.

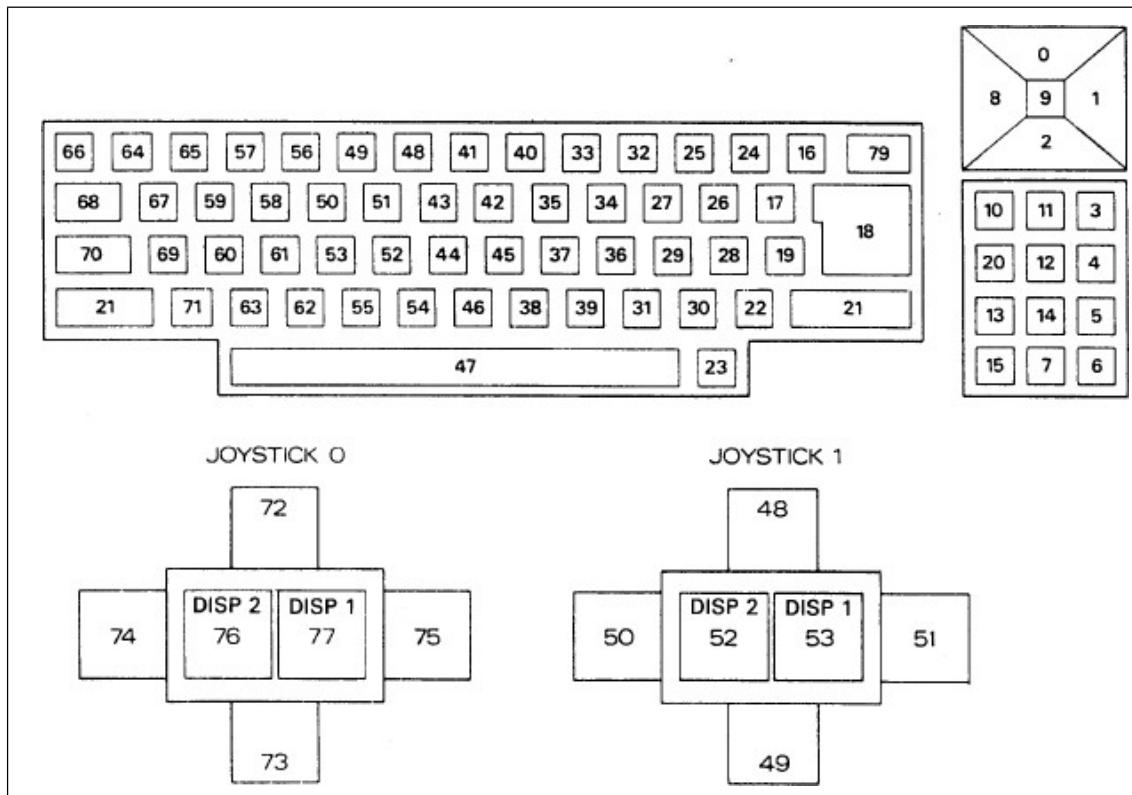


*Fig. 130 C64 palette*

In short, on Amstrad you can choose from more colours and thus always find the right colour for blurring, shading or simply finding the colour tone you need. Amstrad's great success was to create a palette of 27 colours, although you can only use 16 colours at a time. You can also choose 16 colours that don't match each other well and get very tacky graphics (which is impossible on the C64, as you can't choose).

The palette of 27 allowed many Amstrad load graphics to be true works of art.

## 27 APPENDIX III: INKEY codes



Whenever you want to read the keyboard, try first to clear the read buffer of the last keystrokes. It is very common that in a screen where you ask for the user's name for having obtained a high score, the last keystrokes of the game (movements and shots) "sneak in", showing things like "OPPPOQQAAA" in the INPUT command. To avoid this, run something like:

```
10 B$=INKEY$: IF B$<>"" THEN 10
20 INPUT "name:";$name
```

In addition to this recommendation, remember that the fastest way to process the keyboard is as follows:

```
10 IF INKEY(keycode) THEN 30: REM jumps to 30 if not pressed
20 <instructions in case keycode is pressed> 30
```



## 29 APPENDIX IV: AMSTRAD CPC ASCII Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	□	P	^	p	.	^	α	/	—	Ω	↑			
1	Γ	Φ	!	1	A	Q	a	q	■	!	β	^	I	Θ	↓	
2	└	Ω	"	2	B	R	b	r	■	-	„	„	—	Φ	←	
3	└	Ω	#	3	C	S	c	s	■	£	€	„		♦	→	
4	◊	Φ	*	4	D	T	d	t	.	®,	€	^	█	♥	▲	
5	☒	×	5	E	U	e	u	■	I	π	Θ	)	▀	♣	▼	
6	✓	Π	&	6	F	V	f	v	■	r	§	λ	√	○	►	
7	Ω	—	'	7	G	W	g	w	■	†	‘	ρ	⟨	◀	●	
8	←	X	(	8	H	X	h	x	.	-	14	π	✓	❀	□	
9	→	†	)	9	I	Y	i	y	■	12	σ	ς	▀	■	△	
A	↓	Ω	*	:	J	Z	j	z	■	-	34	∅	∞	δ	✖	
B	↑	Θ	+	;	K	[	k	€	■	±	≠	X	▀	♀	✖	
C	Ψ	¶	,	<	L	\	l	l	■	¬	÷	X	/	▀	J	
D	←	▀	-	=	M	]	m	»	■	†	¬	ω	\	▀	▀	
E	Ω	▀	.	>	N	†	n	~	■	†	δ	Σ	▀	▀	‡	
F	Θ	¶	/	?	O	_	o	▀	■	+	i	Ω	▀	▀	+	

	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 APPENDIX V: some sound effects

First of all, you must know that the Amstrad has 3 channels and their identifiers are 1, 2 and 4. To make two channels or all 3 at the same time sound, you just have to add them together

### Use of the SOUND command:

SOUND channel, pitch, duration, volume, envelope pitch, envelope volume, noise

**IMPORTANT:** The volume ranges from 0 to 7 on the CPC464 and from 0 to 15 on the 6128. On the CPC464 the values 8..15 can be used but they are a repetition of the values 0..7 (i.e. 8 means volume 0). This is a BASIC difference between the two models. As a consequence a volume 10 on the CPC6128 is high while on the 464 it is very low.

The noise parameter ranges from 0 to 31

### Examples:

SOUND 1,2000,10,7 : sounds channel 1

SOUND 1+2,2000,10,7 : channels 1 and 2 are sounding

Here are some examples to use directly in your programmes or to inspire you to create other sounds.

Collect a diamond or an ENT coin 1,10,-100,3: sound 1,638,30,30,15,15,0,1	you have been hit by a stone or a projectile ENT 1.10, 100.3: sound 1,638,30,30,15,15,0,1
Boing ENV 1,1,15,1,15,-1,1: SOUND 1,638,0,0,0,1	Boing 2 ENT 2,20,-125,1: SOUND 1,1500,10,12,12,,2
Death ENT 3,100,5,3: SOUND 1,142,100,15,0,3	
Explosion SOUND 7,1000,20,20,15,,,15	Explosion 2 ENV 1,11,-1,25:ENT 1,9,49,5,9,-10,15 SOUND 3,145,300,12,1,1,1,12
Firing ENT -5,7,10,1,1,7,-10,1: SOUND 1,25,20,12,12,,5	



## 31 APPENDIX VI: Interesting firmware routines

In this section I am going to include some firmware routines that can be invoked from BASIC and that can be interesting in your programs.

**CALL 0** : resets the computer

**CALL &bc02** : initialises the palette to its default value. It is a good practice to call it at the beginning of your program in case it is altered.

**CALL &bd19** : synchronise with the screen sweep. If you handle very few sprites you can get a smoother movement, but keep in mind that this instruction will slow down your program a lot.

**CALL &bb48** : disables the BREAK mechanism, preventing the program from stopping if it is running.

**CALL &bd21 , &bd22, &bd23, &bd24, &bd25** : produces a flash effect on the screen

To reset the TIMER of the AMSTRAD:

**On a 6128**

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

**In a 464**

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

To differentiate which machine your programme is on, you must deactivate the music and consult an address with PEEK.

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

**CALL &bca7** : stop playing any sound that was playing.

The GRAPHICS PAPER command exists in CPC6128 but not in CPC464. However, there is a way to have it, using the **CALL &BBE4** and as many parameters with value 1 as the ink colour we want, for example:

**CALL &BBE4,1,1**: same as "GRAPHICS PAPER 2" but works on cpc464

**CALL &BB18** : waits for you to press a key



## 32 APPENDIX VII: Table of Sprite Attributes

The following table contains the memory addresses where the attributes of each sprite are stored, and the length in bytes of each one.

	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

Table 7 Sprite attribute addresses

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
--------------------------	--------------------------	---------------------------	---------------------------	--------------------------	--------------------------	------------------------	-------------------------------

Table 8 flags in the status byte



## 33 APPENDIX VIII: Memory map of 8BP

```
AMSTRAD CPC464 MEMORY MAP of 8BP
;
; &FFFF +-----+
; | display + 8 hidden segments of 48bytes each
; &C000 +-----+
; | system (redefineable symbols, stack pointer, etc.)
; 42619 +-----+
; | bank of 40 stars (from 42540 to 42619 = 80bytes)
; 42540 +-----+
; | character layout map (25x20 =500 bytes)
; | and world map (up to 82 items fit in 500 bytes)
; | Both are stored in the same memory area.
; | because you either use one or you use the other.
; 42040 +-----+
; | sprites (almost 8.5KB for
; | drawings) have 8440 bytes if there are no sequences and no
; | routes)
; +-----+ alphabet images are also stored here.
; | route definitions (of variable length each)
; +-----+
; | animation sequences of 8 frames (16 bytes each)
; | and groups of animation sequences (macro-sequences)
; 33600 +-----+
; | songs
; | (1500 Bytes for music edited with WYZtracker 2.0.1.0)
; 32100 +-----+
; | 8BP routines (8100 bytes or 7100 bytes)
; | here are all the routines and the sprite table
; | includes music player "wyz" 2.0.1.0
; 25000 +-----+
; |
; | YOUR BASIC or CLIST
; | 24KB, 24.8 KB or up to 25KB free for BASIC or C
; | depending on which assembly option you use for 8BP
; |
; |
; 0 +-----+
```



## 34 APPENDIX IX: Available commands 8BP

List of available commands in alphabetical order:

3D, <flag>, #, offsety  3D, 0	Activates the pseudo 3D projection mode.
ANIMA, #	Changes the frame of a sprite according to its sequence
ANIMALL	Changes the frame of sprites with animation flag activated (no need to invoke it, a flag in the PRINTSPALL instruction is enough to invoke it).
AUTO, #	Automatic movement of a sprite according to its Vy,Vx
AUTOALL, <flag routed>, <flag routed>, <flag routed>, <flag routed>.	Movement of all sprites with automatic movement flag active
COLAY, threshold_ascii, @collision, #  COLAY, @collision, #  COLAY, #  COLAY	Detects collision with the layout and returns 1 if there is collision. Accepts a variable number of parameters (always in the same order) from 4 to none.
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%.  COLSP, 32, ini, end  COLSP, 33, @collided%  COLSP, 34, dy, dx  COLSP, #	Returns first sprite with which # collides. The command can be configured with codes 32, 33 and 34.
COLSPALL, @who%, @who%, @withwho%.  COLSPALL, collider  COLSPALL	Returns who collided (collider) and with whom it collided (collided).
LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$, @string\$.	Prints 8x8 image strip and fills map layout
LOCATESP, #, y, x	Changes the coordinates of a sprite (without printing it)
MAP2SP, y, x  MAP2SP, status	Creates sprites to paint the world in scrolling games. Sprites are created with state = status
MOVER, #, dy, dx	relative movement of a single sprite
MOVERALL, dy,dx  MOVERALL	Relative motion of all sprites with relative motion flag active
MUSIC, C, flag, song, speed  MUSIC, flag, song, speed  MUSIC	A melody begins to play. Channel C can be turned off for use with FX effects if desired. No parameters stop ringing
PEEK, dir, @variable%	Reads a 16bit data (can be negative)
POKE, dir, value	enter a 16bit data (which can be negative)
PRINTAT, flag, y, x, @string	Prints a string of redefinable "mini-characters".
PRINTSP, #, y, x  PRINTSP, #  PRINTSP,32, bits	prints a single sprite (# is its number) regardless of status byte. If 32 is specified then we set background bits
PRINTSPALL, ini, fin, anima, sync  PRINTSPALL, ordermode  PRINTSPALL	Prints all sprites with active print flag. If invoked with a single parameter, the ordering mode is set.
RINK,tini,colour1,colour2,...,colourN  RINK, jump	Rotates a set of inks according to a definable pattern composed of any number of inks
ROUTESP, #, steps	Makes you go through N steps of the sprite route at once.
ROUTEALL	Modify sprite speed with path flag (no need to invoke it, just flag in AUTOALL).
SETLIMITS, xmin, xmax, ymin, ymax	Defines the game window, where clipping is done.

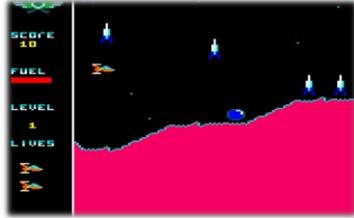
SETUPSP, #, param_number, value	Modifies a parameter of a sprite. If parameter 5 is specified, Vx can optionally be supplied.
STARS, initstar, num, colour, dy, dx	Scroll of a set of stars
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Updates world map items with a subset of elements from a larger map



## 35 APPENDIX X: 8BP Assembly Options

Since version V42, the 8BP library has several assembly options that allow you to choose the capacities you want for your game and thus have more memory available for your game listing.

The assembly option must be specified in the file **Make\_all\_mygame.asm**, which has a specific line to assign the value of the parameter "**ASSEMBLING\_OPTION**".

Option	Description of the option	Example of a typical game
0	You can do any game All commands available You need to use <b>MEMORY 23499</b>  To save library + graphics + music: <b>SAVE "8BP0.bin",b,23500,19119</b>	anyone
1	maze or screen passing games You need to use <b>MEMORY 24999</b> <b>Not available in this mode:</b> <b> MAP2SP,  UMAP,  3D</b>  To save library + graphics + music: <b>SAVE "8BP1.bin",b,25000,17619</b>	
2	For scrolling games You need to use <b>MEMORY 24799</b> Not available in this mode: <b> LAYOUT,  COLAY,  3D</b>  To save library + graphics + music: <b>SAVE "8BP2.bin", b,24800,17819</b>	
3	For games with pseudo 3D You need to use <b>MEMORY 23999</b> Not available in this mode: <b> LAYOUT,  COLAY</b>  To save library + graphics + music: <b>SAVE "8BP3.bin", b,24000,18619</b>	



## 36 APPENDIX XI: RSX/CALL mappings

List of available commands in alphabetical order and their associated address to use the CALL &XXXX invocation when necessary to increase speed. I have given only a few illustrative examples as the usage is identical to the RSX command, except that you must replace the command with CALL <address>.

**IMPORTANT:** From one version of 8BP to another, this list of addresses may vary. Make sure you are using the latest version of 8BP if you are going to use the addresses in this table.

COMMAND	ADDRESS	EXAMPLE
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 &6BD8,0,0,0,0,6
PEEK	&6BB1	CALL &6BB1,dir,@var
POKE	&6BB4	
PRINTAT	&6BC6	CALL &6BC6,0,y,x,@c\$ CALL &6BC6,0,y,x,@c\$
PRINTSP	&6BC3	CALL &6BC3,31
PRINTSPALL	&6B96	CALL &62A6,0,0,0,0,0
RINK	&6BBD	
ROUTESP	&6BCC	
ROUTEALL	&6BD2	
SETLIMITS	&6BDB	
SETUPSP	&6BAB	
STARS	&6BA5	
UMAP	&6BCF	



## 37 APPENDIX XII: 8BP functions in 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 routed>, <flag routed>, <flag routed>, <flag routed>.	void _8BP_autoall(); void _8BP_autoall_1(int flag);
COLAY, threshold_ascii, @collision, #  COLAY, @collision, #  COLAY, #  COLAY	void _8BP_colay_3(int threshold, int* collision, int sp); void _8BP_colay_2(int* collision, int sp); void _8BP_colay_1(int sp); void _8BP_colay();
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%.  COLSP, 32, ini, end  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* collision); void _8BP_colsp_1(int sp);
COLSPALL,@who%,@who%,@with whom%  COLSPALL, collider  COLSPALL	void _8BP_colspall_2(int* collider, int* collided); void _8BP_colspall_1(int collider_ini); void _8BP_colspall();
LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$, @string\$, @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, song, speed  MUSIC, flag, song, 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* data);
POKE, dir, value	void _8BP.poke_2(int address, int data);
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,colour1,colour2,...,colourN  RINK, jump	void _8BP_rink_N(int num_params,int* ink_list); void _8BP_rink_1(int step);
ROUTESP, #, steps	void _8BP_routesp_2(int sp, int steps); 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, value  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, colour, dy, dx	void _8BP_stars_5(int star_ini, int num_stars,int colour, 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 APPENDIX XIII: MiniBASIC in C

BASIC	C prototype
BORDER	<code>void _basic_border(char colour);</code> <code>//example _basic_border(7)</code>
CALL	<code>void _basic_call(unsigned int address);</code> <code>// 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);</code> <code>//takes around 0.3 ms. slow but simple</code>
LOCATE	<code>void _basic_locate(unsigned int x, unsigned int y);</code> <code>// 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);</code> <code>//example: _basic_print("Hello")</code>
RND	<code>unsigned int _basic_rnd(int max);</code> <code>//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);</code> <code>//similar to STR\$</code> <code>//example: _basic_print(_basic_str(num))</code>
TIME	<code>unsigned int _basic_time();</code> <code>//return an unsigned int,(0..65535). As integer, when</code> <code>// reach 32768 go to -32768</code>