

Data structures

Thomas Dettbarn dettus@dettus.net

November 1, 2020

Copyright (c) 2020, Thomas Dettbarn
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The Pawn, The Guild of Thieves, Jinxter, Fish!, Corruption, Myth and Wonderland are interactive fiction games copyright Magnetic Scrolls Ltd, 1984-1990.

Magnetic Scrolls was an interactive fiction developer, based in London UK, active between 1984 and 1990 and pioneer of audio-visually elaborate text adventures.

Contents

1	Introduction	5
1.1	Nomenclature	5
2	Binary File formats	7
2.1	MAG file format	7
2.2	GFX file format, Version 1	7
2.2.1	Symbols in the Huffman Table	8
2.2.2	Bits in the Bitstream	8
2.2.3	Decoding the pixels	8
2.2.4	Rendering the pixels	9
2.3	GFX file format, Version 2	9
2.3.1	16 bit/32 bit	10
2.3.2	Directory	10
2.3.3	Static pictures	10
2.3.4	Animations	11
2.4	DISK1.PIX and DISK2.PIX file format for the MS DOS version	16
2.4.1	Index file	17
2.4.2	DISK1.PIX and DISK2.PIX	17
2.4.3	Layer 3: XOR	19
2.4.4	Translation into pixels	19
2.5	The Pseudo-GFX3 format	20
3	Objects	21
4	Strings	23
4.1	Strings2: The Huffman Table	23
4.2	Nodes	23
5	Dictionary	25
5.1	Plain Dictionary	25

5.2	Packed Dictionary	25
6	Magnetic Windows	27
6.1	Resource files	27
6.2	The directory structure	27
6.3	The game binaries	28
6.4	The graphics	28
6.4.1	Type 7, tree	28
6.4.2	Type 6, animation	29
6.5	Wonderland title screens	29
6.5.1	TITLE.VGA	29
6.5.2	TITLE.EGA	30
6.6	Music	30
7	C64 floppy images	31
7.1	Magnetic Scrolls Directory	31
7.2	Entries in the File list	32
7.3	C64 Pictures	33
7.3.1	Layer 1: Huffman	33
7.3.2	Layer 2: Run Length Encoding	34
7.3.3	The bitmap	35
7.3.4	Colours for The Pawn	36
7.3.5	Colours for Run Length Encoded pictures	36
7.3.6	Colours for Non Run Length Encoded pictures	37
7.3.7	Rendering	38
7.3.8	RGB values	38
7.4	Encryption for the Game code	38
7.4.1	Run Level Encoding	39
7.5	The pseudo .gfx5 format	39
7.5.1	Picture order	40
7.6	The beginning of the Huffman tree	40
8	Atari .STX-Files	41
8.1	The STX file structure	41
8.1.1	The File Header	41
8.1.2	The Track header	41
8.1.3	The sector description	42
8.1.4	The sector payload	42
8.2	The game data	43

8.3	The Game Data G	43
8.4	The Huffman tree H	44
9	Amstrad CPC	45
9.1	DSK format	45
9.1.1	The disk image	46
9.1.2	The file system	47
9.2	Pictures	48
9.2.1	The Index	48
9.2.2	The packed pixel data	48
9.2.3	Rendering the picture	49
9.2.4	The pseudo MaP6 format	51
9.3	Game and String sections in The Pawn	51
9.4	The scrambled sections	51
9.4.1	FILE1, FILE8: Linear scrambled	51
9.4.2	FILE 6: Block scrambling	52

Chapter 1

Introduction

The purpose of this document is to describe the data structures that were used by the original programmers of the Magnetic Scroll Adventures. It is less an exhaustive description, but more of a documentation as on how they were interpreted for the implementation of `dMagnetic`.

1.1 Nomenclature

The Interpreter implemented a virtual 68000 processor. That particular CPU has three data types:

BYTE 8 bits

WORD 16 bits

LONG 32 bits

Unless otherwise stated, all values are stored as BIG endian, meaning that higher bits are stored at a lower address. The value `0x01020304` is thus stored in 4 consecutive bytes as 01 02 03 04.

Chapter 2

Binary File formats

2.1 MAG file format

The game itself is stored in a file with the ending `.mag`. It has the following structure:

Bytes	Description
0..3	"MaSc", the magic header
4..7	Size of all the sections (the whole file)
8..11	Size of the header (=42 Bytes)
13	Version. 0=The Pawn 1= The Guild of Thieves. 2= Jinxter 3= Corruption, Fish 4= Converted from the Magnetic Windows System
14..17	Size of the Game code
18..21	Size of the String 1 section
22..25	Size of the String 2 section
26..29	Size of the Dictionary section
30..33	Pointer to the beginning of the huffman tree
34..37	Size of the Undo(??) section
38..41	Undo PC(??)

Afterwards the Code, String, Dictionary, Dec, Undo Sections follow.

2.2 GFX file format, Version 1

Bytes	Description
0..3	"MaPi", the magic header
4..7	The size of the whole file
8..12	Indexpointer to Picture 0
13..15	Indexpointer to Picture 1
...	...

At the byte that the pointer is denoting, the picture itself is stored as such:

Bytes	Bits	Description
0..1		UNKNOWN
2..3		X1
4..5		X2. the width is X2-X1. <i>width</i>
6..7		height
8..27		UNKNOWN
28..29		RGB value for pixel=0. <i>pal</i>₀
	11..8	red
	7..4	green
	3..0	blue
31..34		RGB value for pixel=1. <i>pal</i>₁
..		
58..59		RGB value for pixel=15. <i>pal</i>₁₅
60..61		Size of the Huffman table (in bytes) <i>HT</i>
62..65		Size of the Data bit stream (in bytes) <i>BS</i>

Afterwards, a Huffman table follows. Then a section of bit streams.

2.2.1 Symbols in the Huffman Table

Symbols in the Huffman Table are either non-terminal symbols, pointing to the next entry, or terminal ones. Terminal entries have bit 7 set.

7	6	5	4	3	2	1	0
0	index pointer						
1	pixel						

The data structure is a tree. Decoding the table starts at the very last one, at Byte $66 + (HT - 1) = ptr$.

2.2.2 Bits in the Bitstream

Decoding of the Pixels is MSB first, so it starts at Byte $66 + HT + BS$ with Bit 7 at the beginning of Bitstream block. If it is a 1, the entry in the Huffman Table $h(ptr) = e$ is being evaluated, otherwise $h(ptr + 1) = e$.

2.2.3 Decoding the pixels

If the retrieved entry e is a non terminal symbol, the new pointer ptr is evaluated as $ptr' = 66 + 2 \cdot e$.

If it is terminal symbol, and $pixel(e) < 16$, the pixel has been decoded as $p_j = pixel(e)$. The pointer is reset to the end of the Huffman table $ptr' = 66 + HT - 1$

If it is terminal symbol, and $pixel(e) \geq 16$, the previous pixel is being used

again, $p_j = p_{j-1}$. This is being repeated $pixel(e) - 15$ times. The pointer is being reset to the end of the Huffman Table $ptr' = 66 + HT - 1$.

Once all the pixels in a line have been decoded, they are being XORed with the previous line:

$$p'_j = p_j \oplus p_{j-width}$$

2.2.4 Rendering the pixels

The RGB values for the pixel are stored in the palette. To render it properly, each pixels RGB value can be drawn as

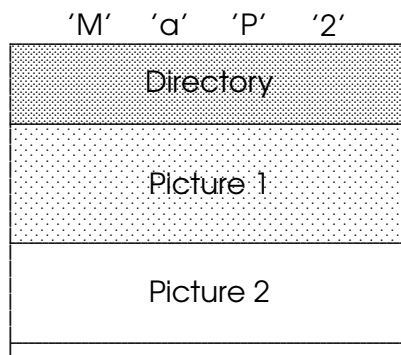
$$rgb_j = pal(p'_j)$$

It should be noted that even tough the entries in the palette are 12 bits wide, the red, green and blue values are only $\in [0..7]$.

2.3 GFX file format, Version 2

The CD collection games, and Wonderland, used a different format for storing pictures.

The .gfx files start with a directory, afterwards the images or animations are being stored as a bitmap-oriented structure.



**IT SHOULD ALSO BE NOTED THAT THE DATA FORMAT IS SOMETIMES LITTLE-
ENDIAN INSTEAD OF BIG ENDIAN!**

2.3.1 16 bit/32 bit

16 bit values are stored as little endians, some 32 bit values are stored as **MIXED** endians.

Bytes	Hex value	Dec value	
little endian	12 3E	0x3e12	15890
mixed endian	1A 2B 3C 4D	0x2b1a4d3c	723143996
BIG endian	5A 6B 7C 8D	0x5a6b7c8d	1516993677

2.3.2 Directory

After the magic header "MaP2", the size of the directory is stored as 16 bit **little endian**. Entries are 16 bytes long.

Bytes	Description
0..3	"MaP2"
4..5	Length of the Directory (in bytes), BIG ENDIAN
6..21	Entry 1
6..13	Filename (case insensitive, zero-terminated)
14..17	Offset within the file (BIG ENDIAN)
18..21	Length of the picture in bytes (BIG ENDIAN)
22..37	Entry 2
...	

2.3.3 Static pictures

Once the filename has been resolved, the size of the picture in bytes, as well as its offset is known.

Bytes	Description
offset+4..offset+5	RGB(0) RGB value pixel 0 (little endian)
offset+6..offset+7	RGB(1) RGB value pixel 1 (little endian)
....	
offset+36..offset+37	RGB(15) RGB value pixel 15 (little endian)
38..41	<i>datasize</i> of the bitmap in bytes, MIXED ENDIAN
42..43	<i>width</i> in pixels, little endian
44..45	<i>height</i> in pixels, little endian
46..47	UNKNOWN
48..47+datasize	bitmap
48+datasize..49+datasize	"D0 5E" identifies static pictures.

RGB values

RGB values are stored as 12 bits in a 16 bit little endian value. The bytes 53 01 become the value 0x0153, meaning RED=1, GREEN=5, BLUE=3.

Bitmap

The Bitmap is organized in lines. Each pixel 0..15 can be represented by 4 bits: 3210. In each line, the bits are lumped together, beginning with bit 0 of the first pixel. Then Bit 0 of the second pixel, then bit 0 of the third and so on. (MSB first).

The bit groups are byte aligned; when the number of pixel in each line is NOT divisible by 8, the lower bits of the last byte are padding. Afterwards, the block for bit 1 starts. Then padding, then Bit 2, then padding, then Bit 3.

The Bitmap for a picture that is 5 pixels wide and 4 pixel requires 16 Bytes:

00000	ppp	11111	ppp	22222	ppp	33333	ppp	Line 1
00000	ppp	11111	ppp	22222	ppp	33333	ppp	Line 2
00000	ppp	11111	ppp	22222	ppp	33333	ppp	Line 3
00000	ppp	11111	ppp	22222	ppp	33333	ppp	Line 4

In this example: *datasize* = 16, *width* = 5 and *height* = 4.

Decoding and rendering

To decode a pixel p , the bits b_0 , b_1 , b_2 , b_3 have to be combined:

$$p_j = 1 \cdot b_0(j) + 2 \cdot b_1(j) + 4 \cdot b_2(j) + 8 \cdot b_3(j)$$

The resulting p_j is the index pointer into the RGB table.

$$rgb(j) = RGB(p_j)$$

Obviously, $rgb_j = 0x000$ is black, $rgb_j = 0x777$ is bright white. $0x700$ is bright red, $0x030$ medium green, $0x001$ is dark blue.

2.3.4 Animations

Animations consist of a background picture, a number of animation "cels", a positioning table for moving objects and a command sequence.

Background
Cels
Animation objects
Commands

The idea is, that one command selects a number of animation objects that are being triggered. Another commands defines how many frames are rendered with the selected objects. Each step of the animation has an X/Y coordinate and a cel number, to select the one to be drawn on top of the background picture.

Background picture

The format of the background picture is the same as the one for static pictures, except for the last two bytes.

Bytes	Description	
offset+5..offset+5	RGB(0)	RGB value pixel 0 (little endian)
offset+6..offset+7	RGB(1)	RGB value pixel 1 (little endian)
....		
offset+36..offset+37	RGB(15)	RGB value pixel 15 (little endian)
38..41	<i>datasize</i>	of the bitmap in bytes, MIXED ENDIAN
42..43	<i>width</i>	in pixels, little endian
44..45	<i>height</i>	in pixels, little endian
46..47		UNKNOWN
48..47+datasize	bitmap	
48+datasize..49+datasize	"00 00"??	identifies the picture as background picture.
50+datasize..51+datasize		UNKNOWN

Animation cels

The cels are sometimes transparent pictures that share their palette with the background picture.

Number of cels
Cel 1
Cel 2
Transparency 2
Cel 3

Obviously, the cel block starts at $offset + 50 + datasize$.

Bytes		Description	
0..1		Number of Cels (Little endian)	
2..5	$datasize'$	$datasize$ cel 1 in bytes (mixed endian)	The
6..7	$width'$	width of cel 1 in pixels (little endian)	
8..9	$height'$	height of cel 1 in pixels (little endian)	
10.. $datasize' + 11$		Bitmap	
$datasize' + 12$	$widthT$	Width of the Transparency Mask	
$datasize' + 14$	$heightT$	Height of the Transparency mask	

bitmap format is the same as for the static images. The RGB values are the same as the background picture's.

When the cel is transparent, $widthT = width'$ and $heightT = height'$, otherwise UNKNOWN.

Transparency

Bytes		Description
$datasize' + 12$	$widthT$	Width of the Transparency Mask (little endian)
$datasize' + 14$	$heightT$	Height of the Transparency mask (little endian)
$datasize' + 16$	$sizeT$	Size of the transparency mask in bytes (little endian)

When the animation cel is transparent, the transparent pixels are marked by a "1" in the transparency mask. The format is MSB first.

When the amount of pixels in the cel picture is not divisible by 8, the last bits of a line are padding.

For a cel with 5 pixels width and 4 lines height=20 pixels, the transparency mask occupies 4 bytes:

ttttt	ppp	Line 1
ttttt	ppp	Line 2
ttttt	ppp	Line 3
ttttt	ppp	Line 4

whereas for a cel with 3 pixels width and 8 lines height=24 pixels, the transparency mask occupies 3 bytes:

ttt	Line 1
ttt	Line 2
tt t	Line 3
ttt	Line 4
ttt	Line 5
t tt	Line 6
ttt	Line 7
ttt	Line 8

Afterwards, 2 Bytes are UNKNOWN.

Animation steps

Between the cels and the animation steps, 2 bytes are UNKNOWN.

This block contains the animations. Basically, an animation is a list of cel numbers, and where to draw them: Each entry denotes the position and the number of the cel to be drawn within a single frame. The later the animation in the block, the later it is supposed to be drawn in the frame. I.E. it is in the foreground layer.

Bytes	Description
0..1	<i>anim</i> s Number of animations (little endian)
2..3	UNKNOWN
4..5	<i>steps</i> 1 Number of steps for animation 1 (little endian)
6..7	UNKNOWN
8..9	$x_{1,1}$ X-Coordinate for the first step (little endian)
10..11	$y_{1,1}$ Y-Coordinate for the first step (little endian)
12..13	<i>cel</i> _{1,1} Number of the first cel (little endian)
14..15	UNKNOWN
16..17	$x_{1,2}$ X-Coordinate for the second step (little endian)
18..19	$y_{1,2}$ Y-Coordinate for the second step (little endian)
20..21	<i>cel</i> _{1,2} Number of second cel (little endian)
22..23	UNKNOWN
...	
2 bytes	$x_{1,steps1}$ X-Coordinate for the last step (little endian)
2 bytes	$y_{1,steps1}$ Y-Coordinate for the last step (little endian)
2 bytes	<i>cel</i> _{1,steps1} Number of last cel (little endian)
2 bytes	UNKNOWN
2 bytes	<i>steps</i> 2 Number of steps for animation 2 (little endian)
2 bytes	UNKNOWN
2 bytes	$x_{2,1}$ X-Coordinate for the first step (little endian)
2 bytes	$y_{2,1}$ Y-Coordinate for the first step (little endian)
2 bytes	<i>cel</i> _{2,1} Number of last cel (little endian)
2 bytes	UNKNOWN

After the last step in the list, the animation loops back from the beginning. When the number of the cel is = -1, it is an end marker. The animation is no longer being shown.

x and y denote where the *cel* is being drawn. Pixels outside the background image are not being drawn. When the transparency mask has a bit set = 1, the pixel is not being drawn.

THE LAST ANIMATION STEP DOES NOT HAVE THE UNKNOWN VARIABLE!

Commands

There are commands for selecting an animation. A command has up to 3 parameters. Each command and parameter is 1 Byte long.

The command line block begins with the number of commands, stored as a 16 bit signed integer.

Bytes	Description
0..1	Number of commands (little endian).

Afterwards, the commands follow:

Command	Parameters	Description
"0x00"		End Marker
"0x01"	<i>animation, start, count</i>	Select an animation
"0x02"	<i>frames</i>	Render Frames
"0x03"	<i>addr_{lsb}, addr_{msb}</i>	jump to instruction <i>addr</i>
"0x04"	<i>delay_{lsb}, delay_{msb}</i>	pause for <i>delay</i> cycles
"0x05"	<i>chance, addr_{lsb}, addr_{msb}</i>	in 1 in <i>chance</i> , jump to instruction <i>addr</i>
"0x06"	<i>addr_{lsb}, chance_{msb}</i>	jump to <i>addr</i> , if running

Currently, it is unknown if *addr* is the instruction number, or its offset

Command "0x01" is referring to animation cel $x_{animation, start}$. Here, *animation* = 1 is the first animation, *start* = 1 is the first frame in the animation block.

Rendering the animations

The command list is parsed from beginning to end. In case command "0x01" occurs, the animation *animation* is being selected. The first animation step being shown will be *start*. The animation itself will be running for *count* frames. As long as more "0x01" commands occur, other animations are being selected.

Command "0x02" will start the animations. A total of *frames* are being rendered. The first frame will start with the background image. The animations are being drawn one after another. The $cel_{anim, step}$ is drawn at the coordinates $x_{anim, step}$, $y_{anim, step}$.

This is obviously being restricted by the size of the background image. In case the cel defines a transparency mask, this has to be reflected as well.

For the next frame *step* is being increased. When *step* reaches the end of the animation list, *step* loops back to *step* = 1.

In case $cel_{anim, step} == -1$, the animation has ended and should be hidden.

Once all the frames for Command "0x02" have been rendered, the selected animations will no longer be shown. The next command is being parsed, until the last command has been finished.

2.4 DISK1.PIX and DISK2.PIX file format for the MS DOS version

The graphics for the MS DOS version are stored in a total of 3 files: DISK1.PIX, DISK2.PIX and an individual index file, ending with a 4. (PAWN4, GUILD4, JINX4, FILE4, CORR4).

2.4. DISK1.PIX AND DISK2.PIX FILE FORMAT FOR THE MS DOS VERSION 19

Within, images are stored as half-tone images. Meaning, that each pixel is in fact encoding 2 pixels. On a cathode ray screen, this produced the illusion of pictures with a richer amount of colours.

2.4.1 Index file

The index file (ending in a 4), contains the offsets into the DISK1.PIX, DISK2.PIX. It is always 256 bytes in size, and broken down into two sections. The first section of 128 bytes contains 32 values (signed 32 bit, little endian), used as offsets into the DISK1.PIX file. The second section uses the 32 values as offsets into the DISK2.PIX file.

-1	Section 1
18	
-1	
-1	
-1	
5	
13	Section 2
-1	
23	
42	
65	
-1	

To read the offset, the *picnum*-th value is read from both sections. One of them has a -1, the other one has a valid offset.

	Section 1	Section 2
	-1	13
	18	-1
	-1	23
⇒	-1	42
	-1	65

It should be noted that the Title screen is typically picture number 30. If not, it is stored in offset 0 within the DISK1.PIX file.

2.4.2 DISK1.PIX and DISK2.PIX

Images in this format are encoded in three layers: The Huffman layer, the Repetition layer, and the XOR layer.

At the offset read from the index file, the image starts with a Huffman table.

Bytes	Description
0	h Length of the Huffman tree (in bytes)
1.. h	H Huffman tree
$h + 1$.. $h + 2$	u Unpacked size (16 bit, big endian) in words
$h + 3$..??	Bitstream

The actual size of the unpacked, “unhuffed” buffer is given in 32 bit words. The size in bytes is $4 * u + 3$. Information like the rgb values, the height and the width is part of the unhuffed buffer.

Layer 1: Huffman

The Huffman tree is being read from the beginning, i.e. byte 1. The entries in the Huffman tree are either nodes or leaves. They are stored in pairs of two bytes. If the bitstream (which is being read MSB first) has a bit set, the left byte is being evaluated. Otherwise the right one.

In case the evaluated byte has bit 7 set, it is a leaf. The terminal symbol can be extracted by removing this bit. Thus, terminal symbols in the “unhuffed” buffer will only be 7 bits wide.

Otherwise it is a node, a link to the next entry within the tree: To translate it into a byte address a , the calculation $a = 2n_j + 1$ has to be performed. Then b_{a+0} will be the next left node, and b_{a+1} will be the next right entry. Once the image has been “unhuffed”, the data starts with a header.

Bytes	Description
0	“0x77” A magic marker
1	m The number of half tone pixels called “stipples”
2..3	w Width (2x6 bit Big endian)
4..5	h Height (2x6 bit Big endian)
6..21	$rgb(0), \dots, rgb(15)$ RGB values (3x2 bit)
$22..22 + 2 \cdot m$	Stipple translation table
$22..22 + m$	p_l Left pixels
$22 + m..22 + 2 \cdot m$	p_r right pixels
$23 + 2 \cdot m..23 + 2 \cdot m + x$	S Stipple string

w and h are stored as 2x6 bit big endian values, since terminal symbols in the Huffman tree can only be 7 bits wide. To translate them into “real” values can be done by

$$\begin{aligned} w &= b_2 * 64 + b_3 \\ h &= b_4 * 64 + b_5 \end{aligned}$$

The rgb values are stored as, MSB first: 2 bits 00, 2 bits red, 2 bits green and 2 bits blue.

Thus, 0x00 is black, 0x3f is bright white, 0x30 is bright red.

Layer 2: Repetitions

The stipple image is a string of stipples

$$S = \{s_0, \dots, s_j, \dots, s_x\}$$

This will be translated into

$$T = \{t_0, \dots, t_k, \dots, t_y\}$$

with $x \leq y$ and $y = w \cdot h - 1$.

Each $s_j \in S$ can be one of three cases:

- $s_j < m$ is a terminal stipple. $t_k = s_j$
- $s_j = m$ and $s_{j-1} \neq m$ is a very special character. s_j will be ignored, but s_{j+1} will be used verbatim. $t_k = s_{j+1}$
- $s_j > m$ and $s_{j-1} \neq m$ is a special character, denoting a repetition of the previous stipple. $t_k, \dots, t_{k+s_j-m-1} = t_{k-1}$

2.4.3 Layer 3: XOR

The translated image T has the dimensions of the final image. However, it has to be XORed over two lines. This extra step resulted in a better packing ratio for the half tone images.

$$u_k = \begin{cases} t_k \oplus t_{k-2 \cdot w} & \text{when } k \geq 2 \cdot w \\ t_k & \text{when } k < 2 \cdot w \end{cases}$$

2.4.4 Translation into pixels

The image is a halftone image. Meaning, that each u_k is actually encoding 2 rgb values. They can be restored from the stipple translation tables p_l for the left and p_r for the right pixel by

$$\begin{aligned} c_l &= \text{rgb}[p_l(u_k)] \\ c_r &= \text{rgb}[p_r(u_k)] \end{aligned}$$

2.5 The Pseudo-GFX3 format

Internally, dMagnetic is using a MaP3 format to combine the index file and the DISK1.PIX and DISK2.PIX in a single buffer.

The buffer starts with the magic word "MaP3". Then there is 4 bytes BIG endian for the length of the Index section (always =256).

Afterwards, 4 bytes of length for the DISK1.PIX file (BIG endian).

Afterwards, 4 bytes of length for the DISK2.PIX file (BIG endian).

Then the Index file.

Then the DISK1.PIX file.

Then the DISK2.PIX file.

Chapter 3

Objects

Objects are stored in a 14 byte structure.

Bytes	Bits	Description
0..4	0..39	UNKNOWN
5	7..1	UNKNOWN
6	0	is described
	7	worn
	6	bodypart
	5..4	UNKNOWN
	3	room
8..9	2	hidden
		parent object
		UNKNOWN
10..13		

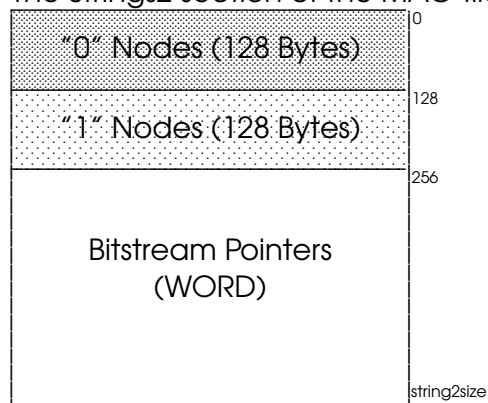
Chapter 4

Strings

Strings are Huffman-Coded. The Decoding table is stored in the Strings2 section of the Mag-File. The Bitstreams are stored in the Strings1 section.

4.1 Strings2: The Huffman Table

The Strings2 section of the MAG-file has the following structure:



The first 256 Bytes are reserved for the Nodes of the Huffman table. Afterwards, the Bitstream pointers, stored as 16 bit WORD values. They are denoting the start index within the Bitstream in the Strings1 section.

4.2 Nodes

The First 256 Bytes contain the Nodes for the Huffman Table. There are two types of nodes: Non-terminal and terminal ones. Terminal nodes have the highest Bit 7 set.

7	6	5	4	3	2	1	0	The index pointers
0	node pointer							
1	symbol							

Chapter 5

Dictionary

5.1 Plain Dictionary

The dictionary contains the names of the objects. Most of the time, objects are using a single word. Since version 1(2?), some objects can be multiple words, such as "can of worms" or "one ferg".

The letters of the word are a..z, the end of a word is marked with Bit 7 being set.

In addition to this, the dictionary itself is broken down into banks. Two banks are searated by 0x82. The end of the dictionary is marked by a 0x81.

Version 4 saw the introduction of 0xA0. But its role is unclear to me.

5.2 Packed Dictionary

The MS DOS versions of Jinxter, Fish and Corruption packed the dictionary in a Huffman tree.

The file is as followed:

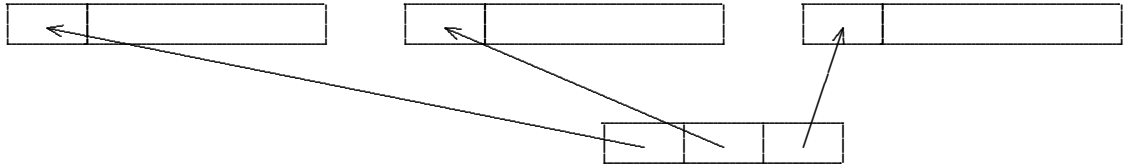
Bytes		Description
0	h	The size of the Huffman table in bytes
1... h	b	The branches of the tree
$h + 1, h + 2$		UNKNOWN
$h + 3, \dots$		The bitstream

The branches in the tree are either nodes or leaves (terminal symbols.)
The terminal symbols are signalled by having bit 7 set.

The tree starts at the beginning. $j = 0$. The bitstream is read MSB first. If the bit is set, b_{2j+0} is followed. Otherwise b_{2j+1} . In case it is a node, $j' = b$. In case Bit 7 of b is set, a leaf has been reached.

Within the tree, the terminal symbols are only 6 bits wide. To transform this into the Plain Dictionary (chapter 5.1), 4 symbols are combined into

3; the fourth symbol contains the 2 MSB from the previous 3 symbols.



Chapter 6

Magnetic Windows

Wonderland, and the Magnetic Scrolls Collection were published using the Magnetic Windows system. This system is combining smaller resource files into a larger files. **If not otherwise stated, numbers are stored as little endian.**

6.1 Resource files

The resource files are named ONE.RSC, TWO.RSC, THREE.RSC, FOUR.RSC, FIVE.RSC, SIX.RSC and SEVEN.RSC. For "The Guild of Thieves", they are given the prefix G. For "Corruption", they get a C. "Fish!" is prefixed by an F.

The files can be seen as one large file.

ONE	TWO	THREE	FOUR	SIX	SEVEN
-----	-----	-------	------	-----	-------

The first 4 bytes are a 32 bit little endian is a pointer p to the directory within this large file.

6.2 The directory structure

Beginning at the byte p , the directory starts. The Files $k = 0, \dots, n - 1$.

Bytes	Bits	Description
p	16	Number of entries n .
$p + 18k + 2$	16	UNKNOWN
$p + 18k + 4$	32	Offset o
$p + 18k + 8$	32	Length l
$p + 18k + 12$	48	Name
$p + 18k + 18$	16	Type t

The types are one of the following

Type	Description
0	Void
1	Tandy
2	WildCard
3	Text
4	Binary
5	Bitmap
6	Animation
7	Tree
8	Font
9	SBinary
10	Cursor
11	VGA
12	EGA

6.3 The game binaries

The game binaries are type 4, Binary. They are called wtab, text, code and index. They are given one of the c,f and g as a prefix.

6.4 The graphics

Graphics are spread out over two types. Type 7, tree contains the Huffman tree. Type 6, Animation, contains the Bitstream, the palette, the height and the width information.

6.4.1 Type 7, tree

The Huffman tree stores the branches and the terminal symbols as 9 bit words. Those 9 bits are split up into two sections.

32 Byte Terminal bitmask. Read MSB first. when a bit is set, it denotes a terminal symbol.

256*1 Byte branch. If the corresponding bit is set, it is a terminal symbol. Otherwise a branch, a link to the next branch.

If the bit from the bitstream is set, the right branch is followed (bitmask: 0..0x1f. branch: 0x20...0x11f). Other wise the left branch (bitmask:0x120..0x13f. branch: 0x140...0x23f).

The byte 0x240 (=576) is the escape character, used in the run level encoding.

6.4.2 Type 6, animation

IT SHOULD BE NOTED THAT THE BITSTREAM IS LONGER THAN THE ACTUAL PICTURE. (Due to a bug in the original encoder) 4 byte magic

16*2 byte RGB. (0x0rgb)

2 byte width

2 byte height

2 byte transparency color

2 byte size s

s byte bitstream.

The bitstream is being read MSB first.

After the Huffman decoding has been finished, the picture contains loops. A loop starts with the escape character.

escape + 0xff = escape character

escape + r + XX = the character XX is being repeated $r + 4$ times

Once the loops have been unrolled, each line is XORed with the previous one.

Once this has been done, the nibbles need to be swapped.

6.5 Wonderland title screens

Wonderland has two title screens. One for the VGA mode, one for the EGA mode.

6.5.1 TITLE.VGA

The title screen for the VGA mode closely resembles a binary version of the XPM format, sometimes called QDV.

- 2 Bytes width w (BIG endian)
- 2 Bytes height h (BIG endian)
- 1 Byte amount of colors -1 c
- $3 \cdot (c + 1)$ Bytes palette 8 bit red, 8 bit green, 8 bit blue
- $w \cdot h$ Bytes pixel

6.5.2 TITLE.EGA

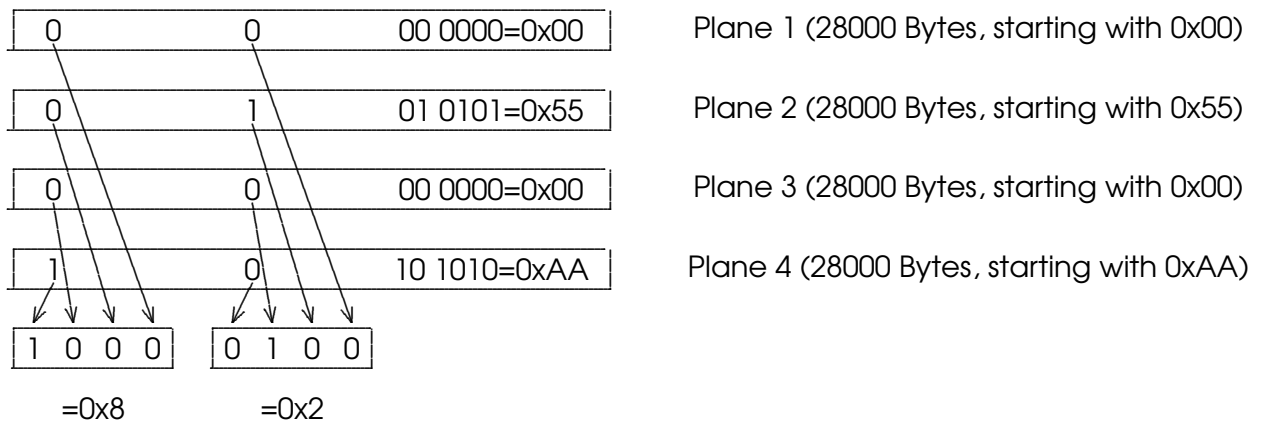
The title screen for the EGA mode has been separated into planes, to work better with the graphics adaptors of the time. Its resolution is 640x350, and 4 bits color depth. This information is NOT PART of the file. Since $640 \cdot 350 / (4 \cdot \frac{8}{4}) = 28000$, each plane is 28000 Bytes wide.

The first 16 Bytes are the palette, 6 bit RGB, 00rrggbb.

Afterwards, the next $28000 \cdot 4 = 112000$ bytes are the bit masks.

To combine the pixel $p[j]$ value from the j th bit within the 4 planes, one has to calculate

$$\begin{aligned} p[j] = & 1 \cdot b[j + 0 \cdot 8 \cdot 28000] + \\ & 2 \cdot b[j + 1 \cdot 8 \cdot 28000] + \\ & 4 \cdot b[j + 2 \cdot 8 \cdot 28000] + \\ & 8 \cdot b[j + 3 \cdot 8 \cdot 28000] \end{aligned}$$



In this example, the planes are being read MSB first. For rendering the first 8 pixels, the RGB values from Byte 8 and Byte 2 are being chosen.

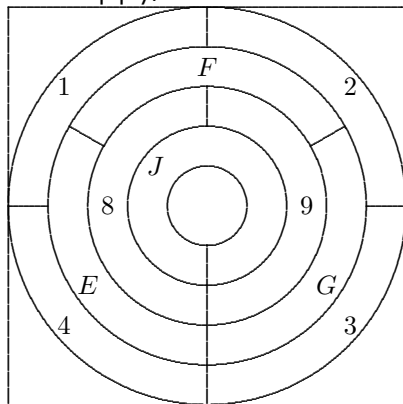
6.6 Music

The Music for Wonderland is stored in the files "t-cat", "t-croq", "t-crt", "t-madt", "t-mus", "t-pal" and can be played back as Standard MIDI data. (With timidity, for example).

Chapter 7

C64 floppy images

The .d64 files are a images of Commodore C64 floppy disks. It contains the sectors of the floppy in consecutive order. Each sector is 256 bytes long. They are grouped in tracks. Depending on its location on the original floppy, each track holds between 17 and 21 sectors.



1	2	3	4
E	F	G	8
9	J		

Since on a circular disk, the outer tracks are in fact longer than the inner tracks, the amount of sectors per track is as followed:

Tracks	Number of sectors	Total amount	Offset
1-17	21	357	0x00000
18-24	19	133	0x16500
25-30	18	108	0x1EA00
31-35	17	85	0x25600

Magnetic Scrolls floppies have 35 tracks. Track 18 is the default directory track. Large data blocks, starting on Track 17, are being continued on Track 19.

7.1 Magnetic Scrolls Directory

The Magnetic Scrolls games use an efficient datastructure for the game data. Its header is stored in Track 1. The sectors contain the following information:

Sector	Contains
0	UNKNOWN
1	A Magic word, for each game (among other things)
2	A list of file pointers

The magic word can be used to detect the game.

Magic word	Game
ARSE	Jinxter
COKE	Corruption
GLUG	Fish!
GODS	Myth
PAWN	The Pawn
SWAG	The Guild of Thieves

The list of entries can be used to find the *Files*. Each entry is 4 bytes long. So the maximum number of entries in the list is 64.

Each entry is 4 bytes long:

Byte	Purpose	Comment
0	Track number	=1: First Track
1	Sector number	=0: First Sector
2	Length (in sectors)	=0: Unused entry
3	Side	=0: Both sides =1/2: only one side of the floppy (see below)

There are instances when the length of the file is not consistent with the entry in the list.

The "Side" entry is used to distinguish between the front side of the floppy or the back side. That does not necessarily mean that 1 is the Front and 2 is the Back. It is rather a mechanism to determine the beginning of the list of pictures.

7.2 Entries in the File list

The position for the game data (See chapter 2.1) Code1, Code2, String1 and String2 is fixed. The position of the pictures vary with each game.

Entry number	File name
1	Code 2. Second half of the Code block
2	String1
3	String2
4	"Cameo" file. A set of thumbnails.
5...	Pictures. THE FIRST ENTRY IS NOT THE FIRST PICTURE.
Last entry	Code1. first half of the code block

The first picture (Picture 0) is the first one with Side=1, which might not necessarily be Entry 5.

The String1 and String2 sections are stored the same way as in the .MAG

files. Code 1 and Code 2 are sometimes encrypted and run length encoded.

7.3 C64 Pictures

The pictures for the C64 version have a resolution of 160x152 pixels. The *Files* in the D64 image are a packed Bitmap (6080 Bytes), and colour information (762, 1140 or 1520 Bytes, depending on the format). The first three bytes within the image are always 0x3E 0x82 0x81. The bytes are as followed:

Byte	Value		Description
0	0x3E		UNKNOWN
1	0x82	t_0	Left Branch
2	0x81	t_0	Right Branch
\vdots	\vdots	\vdots	\vdots
127		t_{63}	Left Branch
128		t_{63}	Right Branch
129		b_0	Bit stream, MSB first
130		b_1	
\vdots	\vdots	\vdots	\vdots

7.3.1 Layer 1: Huffman

Decoding of the Bitstream starts with the Huffman Tree at t_0 and b_0 . The bitstream is being read MSB first, so for the bytes 77 5D, the bit sequence would be 0111 0111 0101 1101. **NOTE** that bit streams can cross Track borders, and Track 18 must be skipped. Therefore, after reading sector 21 at the end of track 17, the next sector would be track 19, sector 0.

If the bit is set, the left branch is being followed. If the bit is not set, the right branch.

If the branch has Bit 7 set, it is being followed to the next branch. If bit 7 is 0, it is a leaf, a terminal symbol. The tree is being reset to the first branch t_0 .

In other words: Let $\beta_l(j)$ be the left branch, located at Byte $2 \cdot j + 1$. And $\beta_r(j)$ be the right branch, located at Byte $2 \cdot j + 2$. S_l be the decoded terminal symbols.

Then:

1. $j := 0$, $k := -1$, $l := 1$, $m := 0x0$
2. Shift m right by 1 bit

3. if $m = 0$ then $k := k + 1$, $m := 0x80$
4. if $(b_k \text{ AND } m)$ then $\beta := \beta_l(j)$ else $\beta := \beta_r(j)$
5. if $(\beta \text{ AND } 0x80)$ then $j := \beta \text{ AND } 0x7f$. Goto 2
6. $j := 0$, $S_l := (\beta \text{ AND } 0x3f)$.
7. $l := l + 1$
8. Repeat at 2 until the Huffman tree has been decoded

Terminal symbols are only 6 bits wide, so 4 consecutive terminal symbols $S_j, S_{j+1}, S_{j+2}, S_{j+3}$ are being combined into 3 Bytes B_k, B_{k+1}, B_{k+2} in the following way:

$$\begin{array}{c}
 \begin{array}{|c|c|c|c|}
 \hline
 00AAAAAA & 00BBBBBB & 00CCCCCC & 00aabbcc \\
 \hline
 S_j & S_{j+1} & S_{j+2} & S_{j+3} \\
 \hline
 \end{array} \\
 \Downarrow \\
 \begin{array}{|c|c|c|}
 \hline
 aaAAAAAA & bbBBBBBB & ccCCCCCC \\
 \hline
 B_k & B_{k+1} & B_{k+2} \\
 \hline
 \end{array}
 \end{array}$$

The first two bytes B_0, B_1 have a special meaning:

- In THE PAWN, they are colours
- In any other game, $B_0 = 0$ means that the run length encoding is not being applied. B_1 is the "background" colour.

7.3.2 Layer 2: Run Length Encoding

If the picture is run length encoded, B_2 determines the amount l of Run Length Codes. Bytes $3, \dots, 3+l$ are the codes. Their position is important.

$$\begin{aligned}
 R &= [B_3, \dots, B_{3+m}, \dots, B_{3+l}] \\
 &= [r_1, \dots, r_m, \dots, r_l]
 \end{aligned}$$

If $B_k = r_m$ for any $k > (5 + l)$ occurs, B_{k-1} is being repeated m times:

$$\begin{aligned}
 B_k &= B_{k-1} \\
 B_{k+1} &= B_{k-1} \\
 &\vdots \\
 B_{k+m-1} &= B_{k-1}
 \end{aligned}$$

7.3.3 The bitmap

The bitmap is always 6080 Bytes long. For the Pawn, this was followed by 64 Bytes padding, bringing it up to 6144 Bytes.

- If the file was run length encoded, those are the bytes $B_{4+l}, \dots, B_{6083+l}$.
- If the file was not run length encoded, the bitmap is stored in the bytes B_2, \dots, B_{6081} .

For the sake of simplicity, the β_m is being introduced:

$$\begin{array}{ccc} B_{4+l} & \dots & B_{6083+l} \\ B_2 & \dots & B_{6081} \\ \Downarrow & & \\ \beta_0 & \dots & \beta_{6079} \end{array}$$

Each byte β_m contains information for 4 pixels.

Eight bytes $\beta_m, \dots, \beta_{m+7}$ contain an 4x8 Block. Each 4x8 Block can hold up to 4 colours, by assigning them to pairs of 2 bits. Each m can be translated into coordinates x, y , according to the following formula:

$$\begin{aligned} y(m) &= \left\lfloor \frac{m}{320} \right\rfloor \cdot 8 + m \text{ modulo } 8 \\ x(m) &= \left(\left\lfloor \frac{m}{8} \right\rfloor \text{ modulo } 40 \right) \cdot 4 \end{aligned}$$

Four consecutive pixels $x, x+1, x+2, x+3$ will be given colours determined by pairs of bits in β_m (MSB first):

Bits of β_m	76	54	32	10
x-coordinate	$x(m) + 0$	$x(m) + 1$	$x(m) + 2$	$x(m) + 3$

For example, the bytes 1E 67 56 F2 81 00 EF 55 42 represent the following bitmap:

1E	00	01	11	10	01	00	00	10	42
67	01	10	01	11					
56	01	01	01	10					
F2	11	11	00	10					
81	10	00	00	01					
00	00	00	00	00					
EF	11	10	11	11					
55	01	01	01	01					

7.3.4 Colours for The Pawn

The bitmap is padded by 64 Bytes.

Afterwards, Byte β_{6144} is the beginning of a colour map $\gamma_0, \dots, \gamma_{759}$. One graphic mode on the C64 allowed a 4x8 block to have one of two colours, determined by the byte γ_n , or one of two colours determined by fixed memory locations, 0xD021(?) and 0xD022(?).

The picture *File* contains the content for those memory locations in the first two bytes B_0 and B_1 .

To translate n into the upper left coordinates x, y for a block, the following formula can be used:

$$y(n) = \left\lfloor \frac{n}{40} \right\rfloor \cdot 8$$

$$x(n) = (n \bmod 40) \cdot 4$$

The bit pattern within this block is being translated into a colour by the following table:

Bit pattern	Colour
00	B_0 , Bit 3..0
01	γ_n , Bit 7..4
10	γ_n , Bit 3..0
11	B_1 , Bit 3..0

7.3.5 Colours for Run Length Encoded pictures

All games other than The Pawn used a graphic mode. One where within a 4x8 block, the 4 colours were determined by two bytes γ_n and γ_{n+760} . On top of that, the 64 Bytes padding was no longer used, therefore Byte β_{6080} is the beginning of the color map $\gamma_0, \dots, \gamma_{759}, \gamma_{760}, \dots, \gamma_{1519}$.

To translate n into the upper left coordinates x, y for a block, the following formula can be used:

$$\begin{aligned} y(n) &= \left\lfloor \frac{n}{40} \right\rfloor \cdot 8 \\ x(n) &= (n \bmod 40) \cdot 4 \end{aligned}$$

The bit pattern within this block is being translated into a colour by the following table:

Bit pattern	Colour
00	B_1 , Bit 3..0
01	γ_{n+760} , Bit 7..4
10	γ_{n+760} , Bit 3..0
11	γ_n , Bit 3..0

7.3.6 Colours for Non Run Length Encoded pictures

All games other than The Pawn used a graphic mode. One where within a 4x8 block, the 4 colours were determined by two bytes Γ_p and γ_q .

With $p \in 0, \dots, 379$ and $q \in 0, \dots, 759$.

Byte β_{6080} is the beginning of the color map $\Gamma_0, \dots, \Gamma_{359}, \gamma_0, \dots, \gamma_{759}$.

To translate p and q into the upper left coordinates x, y for a block, the following formula can be used:

$$\begin{aligned} y(q) &= \left\lfloor \frac{q}{40} \right\rfloor \cdot 8 \\ x(q) &= (q \bmod 40) \cdot 4 \\ p_1 &= \left\lfloor \frac{p}{2} \right\rfloor & p_2 &= p \bmod 2 \end{aligned}$$

The bit pattern within this block is being translated into a colour by the following table:

Bit pattern	Colour
00	B_1 , Bit 3..0
01	γ_q , Bit 7..4
10	γ_q , Bit 3..0
11	Γ_{p_1} , Bit 7..4, if $p_2 = 0$
11	Γ_{p_1} , Bit 3..0, if $p_2 = 1$







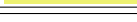




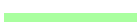



7.3.7 Rendering

Start with $p = 0, m = 0, q = 0$. Top left corner ($x = 0, y = 0$). Determine the four colours, determine the Bit pattern in the 4x8 Bit block. Draw the first four pixels. Then draw the four beneath it. Repeat 8 times. Draw the 4 pixels right of it. Then beneath. Repeat everything 20 times. Then go 8 pixels down. Start at the left side ($x = 0$)...

The last byte from the Bitmap should be at 6080. The resulting image has a resolution of width x height = 160x152.

7.3.8 RGB values

Brix, an expert in C64 programming, suggested the following RGB values for the 16 colours:

Colour	Name	RGB value	
0x0	BLACK	0, 0, 0	
0x1	WHITE	255,255,255	
0x2	RED	129,51,56	
0x3	CYAN	117,206,200	
0x4	PURPLE	142,60,151	
0x5	GREEN	86,172,77	
0x6	BLUE	46,44,155	
0x7	YELLOW	237,241,113	
0x8	ORANGE	142,80,41	
0x9	BROWN	85,56,0	
0xa	LIGHT RED	196,108,113	
0xb	DARK GRAY	74,74,74	
0xc	GRAY	123,123,123	
0xd	LIGHT GREEN	169,255,159	
0xe	LIGHT BLUE	112,109,235	
0xf	LIGHT GRAY	178,178,178	

7.4 Encryption for the Game code

To harden the copyright protection, the game code for the virtual machine uses a simple encryption algorithm.

Each block $B_j = [b_0, \dots, b_p, \dots, b_{255}]$ within the code *File* is SOMETIMES encrypted. The application of the encryption can be determined by looking at the first two bytes of the CODE block. This has to be 0x49 0xFA. Every game starts with those two, they are the equivalent of a LEA instruction.

It can be decrypted with the following algorithm:

1. Select a pivot $p = 0xff \text{ XOR } j \text{ modulo } 8$.
2. If $p \neq 255$: For each $k \in (p + 1), \dots, 255$ perform $b'_k = b_k \text{ XOR } b_p$
3. Afterwards, for each $k \in p - 1, \dots, 0$ perform $b'_k = b_k \text{ XOR } b_{k+(p \text{ XOR } 0xff)}$
4. Finally, revert B_j : For each $k \in 0, \dots, 255$ perform $b'_k = b_{255-k}$

Perform the same operations on the next block B_{j+1} .

7.4.1 Run Level Encoding

To preserve memory, the game code has been packed by run level encoding, but only for bytes having the value 0x00. The first two bytes of the file are the length of the encoded file in BIG endian format. Thus, the first byte b_0 needs to be multiplied by 256 and added to the second byte b_1 .

Afterwards, if the byte $b_j = 0x00$, byte b_{j+1} determines the amount of 0x00.

For example, if the byte sequence is BE 0C 00 03 09, the decoded sequence becomes BE 0C 00 00 00 09.

7.5 The pseudo .gfx5 format

The pseudo .gfx5 format, used internally by dMagnetic has a 133 byte header.

Bytes	Description
0..3	"MaP5" Magic word
4..7	Offset to picture 0 (As the VM expects it, BIG endian)
8..11	Offset to picture 1 (VM view, BIG endian)
⋮	⋮
128..131	Offset to picture 32
132	Version of the Game (0=Pawn)

Starting with Byte 133, the image data follows. It is simply a copy of the sectors of the picture files. They are ordered in the same way they appear in the .d64 image.

1. Side (side 1 first)
2. Track (Track 1 first)
3. Sector (Sector 1 first)

7.5.1 Picture order

Except for Myth, the order of the pictures on the floppy images is different from the ones in other releases, and different from the ones the virtual machine expects. In fact, they have to be reordered.

The actual order is as followed (0=the first image on side 1:)

Jinxter 4, 0, 5, 6, 7, N/A, 8, 1, 9, 10, 11, 12, 13, 14, 15, 16, 17, 2, 3, 27, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27

Corruption 24, 8, 9, 25, 10, 13, 15, 16, 17, 1, 18, 23, 21, 6, 5, 4, 12, 14, 2, 3, 11, 20, 7, 22, 19, 0

Fish! 3, 21, 8, 11, 18, 16, 17, 4, 2, 5, 1, 6, 9, 10, 14, 20, 22, 24, 25, 0, 15, 23, 7, 19, 13, N/A, 26

Myth 0, 1, 2, 3

The Pawn 4, 26, 13, 23, 0, 8, 29, 5, 18, 19, 3, 9, 12, 11, 16, 22, 17, 21, 28, 6, 27, 25, 24, 2, 1, 20, 14, 7, 15, 10

The Guild of Thieves 9, 17, 20, 0, 26, 19, 11, 12, 4, 5, 2, 13, 14, 8, 6, 1, 15, 16, 3, 24, 21, 28, 22, 25, 18, 23, 7, 10, 27

(Some pictures were not available in the C64 release. (N/A))

For example, whilst playing JINXTER, when the virtual machine tries to load picture number 0, it actually has to load the fifth picture that can be found in the .d64 images.

7.6 The beginning of the Huffman tree

The strings are Huffman encoded. The tree to decode it can be found at the following offsets:

Game	Decoding Offset
Jinxter	0x13100
Corruption	0x16100
Fish	0x14e00
Myth	0x08b00
The Pawn	0x0b400
The Guild of Thieves	0x0f100

For "The Pawn" and "The Guild of Thieves", this is the beginning of the second string sections.

It differs for the other games. Here, it can be found by looking for sector borders, since the tree is sector aligned. The previous sector must end with 0x00 0x00 0x00. Each tree in each game starts with 0x01 0x02 0x03.

Chapter 8

Atari .STX-Files

STX is a disk image format which preserved the original structure. This allows for the copyright protection to stay intact, since those sometimes where checking bad sectors and timing offsets when reading tracks from the floppy.

8.1 The STX file structure

I found the description under this link: http://info-coach.fr/atari/documents/_mydoc/Pasti-documentation.pdf.

The numbers are from the intel world, so they are little endian.

8.1.1 The File Header

The files start with a header in the first 16 bytes:

Bytes	Length	symbol	Description
0.. 3	4		0x52 0x53 0x59 0x00="RSY" the magic word
4.. 5	2		Version of the File
6.. 7	2		Tool identifier.
8.. 9	2		Reserved 1
10	1	<i>t</i>	Track Count (mostly 82)
11	1		revision
12..15	4		Reserved 2

8.1.2 The Track header

After the file header, *t* tracks follow. Each track starts out with 16 bytes track header:

Bytes	Length	symbol	Description
0.. 3	4	$r(t)$	record size, the bytesize for the track
4.. 7	4	$f(t)$	size of the fuzzy mask
8.. 9	2	$s(t)$	number of sectors for this track
10..11	2		Flags of the track
12..13	2		Length of the track image
14	1		Track number
15	1		track type

The offset of the track $o(t)$ within the .STX file can be calculated iteratively, with

$$\begin{aligned} o(0) &= 16 \\ o(t) &= o(t-1) + r(t-1) \end{aligned}$$

.STX may contain a fuzzy mask, which has a size of $f(t)$ bytes. The STX files I encountered did not have one.

8.1.3 The sector description

After the track header, at position $o(t) + 16$, the sector headers can be found. There are $s(t)$ many.

For each sector $\sigma \in \{0, \dots, s(t) - 1\}$, the description is 16 bytes long.

Bytes	Length	symbol	Description
0.. 3	4	$d(t, \sigma)$	Data Offset
4.. 5	2		bit position
6.. 7	2		Read time
8	1		ID track
9	1		ID head
10	1		ID number
11	1	$b(t, \sigma)$	ID size (2=512 bytes, 3=1024 bytes)
12..13	2		ID CRC16
14	1		FDC flags
15	1		reserved

In the STX files I encountered, the ID field is not consistent with the description within the STX standard. This might be a copyright protection.

8.1.4 The sector payload

The offset $\omega(t, \sigma)$ of the payload for a sector σ on track t starts at one of

$$\begin{aligned} \omega(t, \sigma) &= o(t) + d(t, \sigma) \\ \omega(t, \sigma) &= o(t) + f(t) + d(t, \sigma) \end{aligned}$$

To me, it is unclear which one of the two is correct.

The amount of bytes $\alpha(t, \sigma)$, however, is mostly either 512 or 1024:

- $\alpha(t, \sigma)=128$, when $b(t, \sigma)=0$
- $\alpha(t, \sigma)=256$, when $b(t, \sigma)=1$
- $\alpha(t, \sigma)=512$, when $b(t, \sigma)=2$
- $\alpha(t, \sigma)=1024$, when $b(t, \sigma)=3$

In other words:

$$\alpha(t, \sigma) = 128 \cdot 2^{b(t, \sigma)}$$

8.2 The game data

The game data is stored in tracks with 1024 byte long sectors. For obscure reasons, the beginning of the game data is actually on the last track, and continues on the former ones. On top of this, the sectors are not being read linearly.

The correct order to read the game data is by calculating the offsets in the following scheme:

$\omega(79, 2)$	$\omega(79, 3)$	$\omega(79, 4)$	$\omega(79, 0)$	$\omega(79, 1)$
$\omega(78, 3)$	$\omega(78, 4)$	$\omega(78, 0)$	$\omega(78, 1)$	$\omega(78, 2)$
$\omega(77, 4)$	$\omega(77, 0)$	$\omega(77, 1)$	$\omega(77, 2)$	$\omega(77, 3)$
$\omega(76, 0)$	$\omega(76, 1)$	$\omega(76, 2)$	$\omega(76, 3)$	$\omega(76, 4)$
$\omega(75, 1)$	$\omega(75, 2)$	$\omega(75, 3)$	$\omega(75, 4)$	$\omega(75, 0)$
$\omega(74, 2)$	$\omega(74, 3)$	$\omega(74, 4)$	$\omega(74, 0)$	$\omega(74, 1)$
$\omega(73, 3)$	$\omega(73, 4)$	$\omega(73, 0)$	$\omega(73, 1)$	$\omega(73, 2)$
\vdots	\vdots	\vdots	\vdots	\vdots
$\omega(1, 4)$	$\omega(1, 0)$	$\omega(1, 1)$	$\omega(1, 2)$	$\omega(1, 3)$

$= G$

Note: The first sector on each track t can be calculated as

$$4 - (t + 3) \bmod 5$$

Track 0 contains the bootloader and a README file.

8.3 The Game Data G

Once the game data has been reordered, the first 256 bytes contain 32 index pointers as BIG endian numbers.

They point to the following positions within *G*:

Number	Bytes	Description
0	0.. 3	UNKNOWN (the unhuffer maybe?)
1	4.. 7	Huffman tree <i>H</i>
2	8..11	Picture 0
3	12..15	Picture 1
⋮	⋮	⋮
31	252..255	Title screen (?)

The pictures are the same format as in the .GFX1 file, sans the header.

8.4 The Huffman tree *H*

Apparently, the Huffman tree contains the game code and the string sections. Each leaf is 4 bytes wide. 2 bytes left, 2 bytes right. Terminal symbols have bit 8 set.

- 0..1 Root index of the tree
- 2..1017 The Huffman tree
- 1030.. The Bitstream

The left branch is at $4*idx+0$, the right branch at $4*idx+2$.

The root of the tree is at the end. The Bitstream is being read MSB first. If the bit is set, the left branch is being followed. Otherwise the right branch.

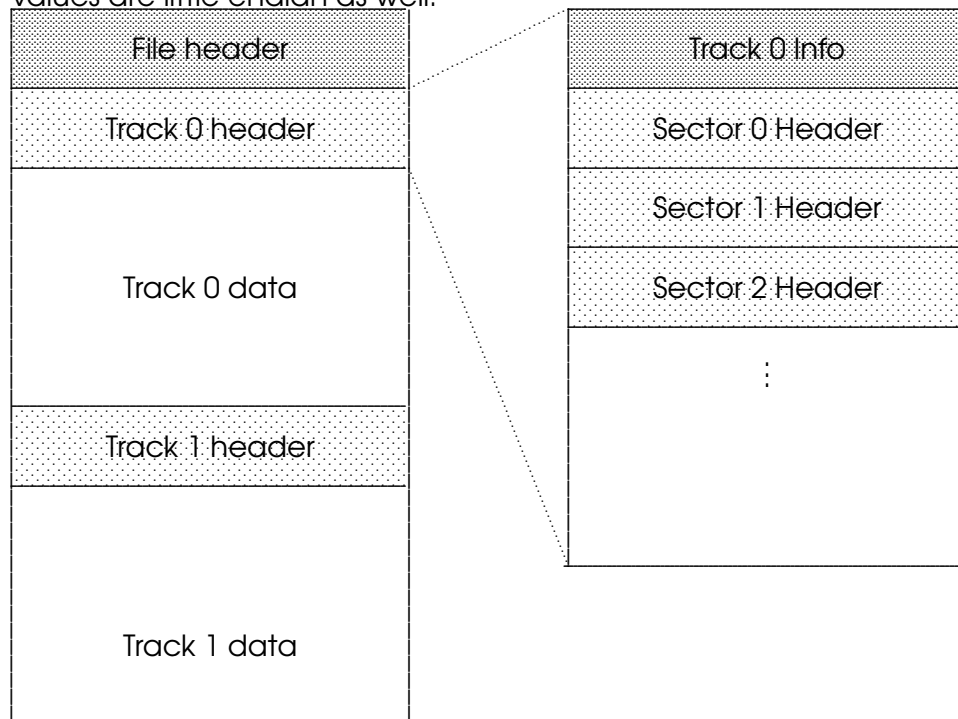
Chapter 9

Amstrad CPC

9.1 DSK format

The standard format to store Amstrad floppy images is the DSK format. Essentially, it is a verbatim copy of the floppy disks, albeit with the sector interleaving intact. Additionally, each track is given a header with details such as the size of each sector etc.

Please note that since the Amstrad CPC is a little endian machine, the values are little endian as well.



The file header and each track header has a size of exactly 256 bytes. After the meaningful data, they are padded with 0.

9.1.1 The disk image

The format of the File header is as followed:

Bytes	Length	Symbol	Description
0..32	33		Magic word
33..47	15		Name of the creator
48	1	n_T	Number of tracks
49	1	n_S	Number of sides
50	1	b_T	Number of bytes in a track data block

With this information, the offset O to the track header track t on the side s can be calculated with the following formula:

$$O(t, s) = 256 + (n_S \cdot t + s) \cdot b_T$$

Bytes	Length	Symbol	Description
0..11	12		Magic word
12..15	4		unused
16	1	t	Track ID
17	1	s	side ID
18..19	2		unused
20	1	β_T	Size indicator for the sectors
21	1	n_σ	Number of sectors
22	1		gap3 length
23	1		Filler byte

The actual number of bytes B for a sector is calculated by

$$B = 2^{\beta_T + 7}$$

The filler byte can be used to detect unformatted sectors.

Afterwards, $n_\sigma \cdot 8$ bytes of sector headers follow:

Bytes	Length	Symbol	Description
0	1		Track number
1	1		Sector number
2	1	$i(\sigma)$	Sector ID, important for deinterleaving
3	1	β_σ	Sector Size, might be different from β_T
4	1		FDC status1
5	1		FDC status2
6..7	2		unused

The sectors are organized the same way they would be on a floppy disk, meaning, they are interleaved. To read the data linearly, they have to be deinterleaved. The actual order of the sectors varies from Disk to Disk, but it can be determined by the sector ID within the sector Headers.

In consecutive order, the track data starts with the sector σ that has the lowest $i(\sigma)$.

The offset to the track data is

$$\Omega(t, s, \sigma) = O(t, s) + 256 + B \cdot \sigma$$

9.1.2 The file system

Once the sector data from the DSK image has been converted into a consecutive, linear image file, it represents a CPM file system. The CPM file systems for Magnetic Scrolls Files have a block size of 1024 bytes. The first 2 blocks are reserved for a directory.

Directory entries are 32 bytes long.

Bytes	Length	Symbol	Description
0	1		UNKNOWN
1..11	11		Filename (8 Bytes+3 Bytes extension)
12..15	4	EX,S1,S2,RC	File size pointers?
16..31	16		16 Block pointers

Multiplying the block pointers by 1024 gives the offset within the file system.

Since a directory entry can address at most 16 blocks, files larger than 16 kByte are being split over multiple entries, all with the same filename. To read the full file, one has to read the blocks in the given order. If the block number is =0, the file is complete.

The filenames can be used to determine the game. It is the same for each file, followed by a number

GAME	PREFIX
The Pawn	PAWN
The Guild of Thieves	GUILD
Jinxter	JINX
Corruption	CORR

So far, I was able to discover the following files and their roles:

PAWN	GUILD	Jinxter	CORR	Description
PAWN0	GUILD0	JINX0	CORR0	The interpreter
PAWN1'	GUILD1*	JINX1*	CORR1*	The code section
PAWN2'	GUILD2	JINX2	CORR2	Second part of the strings
PAWN3	GUILD3	JINX3	CORR3	First part of the strings
PAWN4	GUILD4	JINX4	CORR4	Picture index
	GUILD5	JINX5	CORR5	Pictures on the second disk
	GUILD6*	JINX6*	CORR6*	The code section, continued
	GUILD7	JINX7	CORR7	Pictures on the first disk
		JINX8*	CORR8*	The dictionary

' Huffman coded, see chapter 9.3 for description

* Scrambled, see chapter 9.4 for a description

9.2 Pictures

Note that I wrote this chapter before I had a look at THE PAWN. For this game, the pictures are stored in a single File: PAWN4. It starts with 2 bytes UNKNOWN, the index (with little endian values). Then the Pictures as Tree/Bitstream pairs.

9.2.1 The Index

The index to the pictures can be found in the files ending with 4, so GUILD4, JINX4. The entries are each 4 bytes long: 3 for the offset with either FILE5 or FILE7, and 1 byte to determine which one; if the last byte is =0xff, it is in FILE5. If it is =0x00, it is in FILE7.

9.2.2 The packed pixel data

Once the offset within either FILE5 or FILE7 is known, the pictures are, once again, Huffman-Encoded.

The structure is as followed:

Length	Symbol	Description
1 Byte	t	Length of the tree
$2 \cdot t + 2$ Bytes	T	The Tree
? bytes	B	The Bit stream

Decoding starts with tree index $i = 0$.

The bit stream is being decoded MSB first. In case the bit is set, $t = T(2 \cdot i + 0)$ is being evaluated. Otherwise $t = T(2 \cdot i + 1)$.

Terminal symbols have bit 7 set, so if $t < 128 \Rightarrow i' = t$.

Otherwise, the terminal symbol $\tau = t - 128$.

There are three kinds of terminal symbols: Palette, Codes and Loops.

The first 14 symbols are the palette $\pi_0 \dots \pi_{14}$. They are being used directly, so $\pi_j = \tau$. Afterwards, only codes and loops occur.

The codes make up terminal symbols τ between 0x00 and 0x0f. Loops are between 0x10 and 0x7f. They repeat the last code $\tau - 16$ times.

It takes two codes τ_0 and τ_1 to calculate the actual output byte $b(j)$, by means of a the following codebook:

x	$c(x)$
0	0x00
1	0x40
2	0x04
3	0x44
4	0x10
5	0x50
6	0x14
7	0x54
8	0x01
9	0x41
10	0x05
11	0x45
12	0x11
13	0x51
14	0x15
15	0x55

The formula is

$$b(j) = 2 \cdot c(\tau_0) + c(\tau_1)$$

It can easily be seen that this way all possible values for $b(j)$ can be calculated. Note that in a loop, only τ_1 is being reused.

Once all the bytes $b(0) \dots b(11599)$ have been decoded, they need to be descrambled over 2 lines. Since each line for a picture with a resolution of 160x152 is represented by 80 bytes, the operation would be:

$$\forall_{i=160}^{11599} b'(i) = b(i) \oplus b(i - 160)$$

9.2.3 Rendering the picture

Pixel deinterleaving

The Amstrad used interleaved pixel values. Two pixels are being combined in a byte. To get the pixel value, calculate

```
p0 = (b >> 7) & 0x1 << 0;
p0 |= (b >> 3) & 0x1 << 1;
p0 |= (b >> 5) & 0x1 << 2;
p0 |= (b >> 1) & 0x1 << 3;
```

```
p1 = (b >> 6) & 0x1 << 0;
p1 |= (b >> 2) & 0x1 << 1;
```



























```
p1 |= ( (b>>4) & 0x1) <<2;
p1 |= ( (b>>0) & 0x1) <<3;
```

Those two pixels p_0 and p_1 are translated into the rgb lookup by the following formula

$$r(p_x) = \begin{cases} 0 & : p_x = 0 \\ 26 & : p_x = 1 \\ \pi_{p_x-2} & : otherwise \end{cases}$$

RGB values

The Amstrad CPC had 27 colors to choose from, out of which 16 could be displayed at the same time.

Colour	Name	RGB value	
0	BLACK	0, 0, 0	
1	BLUE	0, 0, 128	
2	BRIGHT BLUE	0, 0, 255	
3	RED	128, 0, 0	
4	MAGENTA	128, 0, 128	
5	MAUVE	128, 0, 255	
6	BRIGHT RED	255, 0, 0	
7	PURPLE	255, 0, 128	
8	BRIGHT MAGENTA	255, 0, 255	
9	GREEN	0, 128, 0	
10	CYAN	0, 128, 128	
11	SKY BLUE	0, 128, 255	
12	YELLOW	128, 128, 0	
13	WHITE	128, 128, 128	
14	PASTEL BLUE	128, 128, 255	
15	ORANGE	255, 128, 0	
16	PINK	255, 128, 128	
17	PASTEL MAGENTA	255, 128, 255	
18	BRIGHT GREEN	0, 255, 0	
19	SEA GREEN	0, 255, 128	
20	BRIGHT CYAN	0, 255, 255	
21	LIME	128, 255, 0	
22	PASTEL GREEN	128, 255, 128	
23	PASTEL CYAN	128, 255, 255	
24	BRIGHT YELLOW	255, 255, 0	
25	PASTEL YELLOW	255, 255, 128	
26	BRIGHT WHITE	255, 255, 255	

The RGB lookupvalue $r(p)$ determines the actual colour of the pixel.

9.2.4 The pseudo MaP6 format

My MaP6 format is as followed: 4 Bytes magic word "MaP6". 32*4 bytes (The Pawn: 29*4 bytes) index pointer (BIG endian). Afterwards the tree and bitstreams for the pictures.

9.3 Game and String sections in The Pawn

The CODE and the String2 sections in The Pawn are being Huffman Encoded. To decode, the tree starts at offset 1, the bit stream (MSB first) at offset 129. Terminal symbols have bit 7 set. Four bytes are being combined into three bytes in the same way as in chapter 5.2.

The game code can be found in the file PAWN1. The first string section is in PAWN3, followed by PAWN2. The images are stored in the file PAWN4.

9.4 The scrambled sections

Starting with releases of "The Guild of Thieves", the game code and the dictionary were scrambled. Since the Amstrad CPC had only a limited amount of memory, some parts of the code were pre-loaded, and the others read from the floppy disks when they were needed. This had an impact on the design of the scrambler.

The first part of the code section was stored in Files ending with 1, the second part in 6. When concatenated, they make up the CODE section of the game.

9.4.1 FILE1, FILE8: Linear scrambled

The code section that starts in FILE1 and the dictionary in FILE8 (when it was available), were treated to the same scrambling with a Pseudo Random Bit Sequence.

The sequence s can be replicated with the following formula:

$$\begin{aligned}\sigma &:= 0x1803 \\ s(0) &= (\sigma + 256 \cdot \sigma + 0x29) \bmod 65536 \\ s(j) &= (s(j-1) + 256 \cdot s(j-1) + 0x29) \bmod 65536\end{aligned}$$

The sequence is being initialized with 0x1803 and continues as 0x1b2c 0x4755 0x9c7e 0x1aa7 0xc1d0.

Obviously, this is a sequence of 16 bit values. To descramble each Byte $b(j)$ with it, the higher and the lower bytes are being xored with it:

$$b'(j) = (s(j) \oplus (s(j) \gg 8) \oplus b(j)) \wedge 0xff$$

9.4.2 FILE 6: Block scrambling

Since the contents of FILE6 are being loaded in a randomized order, it would have been inefficient to scramble them linearly as well. Thus, each block of 128 bytes is given its own PRBS. The starting value σ of the block is identical to the relative offset of the descrambled block within the overall code section.

Thus, when the FILE1 is $0x4000$ bytes in size, the first 128 Bytes are scrambled with $\sigma = 0x4000$, the next one with $\sigma = 0x4080$, afterwards $\sigma = 0x4100$ and so forth. Note that the size of FILE1 changes from game to game.