Decoding Repton

Compiled by Gerald J Holdsworth

Contents

Introduction

Repton has been described as “the thinking man’s arcade game” and was originally published, for the BBC Micro and Acorn Electron, in 1985 by Superior Software. Written by 15-year-old Tim Tyler, this quickly became a smash hit and was followed by Repton 2. Later on, Repton 3 followed, and then Repton Infinity before Repton moved onto the Archimedes with EGO: Repton 4. Recently, Retro Software released Repton: The Lost Realms, with a new set of screens to follow in the near future.

Superior Software, Superior Interactive, Retro Software and Alligata have also published various other versions, for the Acorn Atom, Sinclair Spectrum, Commodore 64, Acorn Archimedes & RISC PC, and Microsoft Windows.

This is a guide to the various formats of file used by the many incarnations of Repton, or the arrangement of the map and sprite data within the code. What it does not cover are the PC Repton formats, except for the graphics. The following are covered:

- BBC Micro Repton, Repton 2, Repton 3 and Repton Infinity
- Acorn Electron Repton, Repton 2, Repton 3 and Repton Infinity
- Sinclair Spectrum Repton and Repton 2
- Commodore 64 Repton 3
- Acorn Archimedes Repton, Repton 2 and Repton 3
- Desktop Repton 1, 2, and 3
- PC Repton 3 graphics

In addition, I have researched and present here some other formats for games similar to the Repton series:

- Ripton (from A&B Computing), which was a direct copy of Repton, for the BBC Micro
- Harry Wood’s Repton 3, which was Harry’s attempt at a PC version of Repton 3, before Superior Interactive produced their’s
- Bonecruncher, a different game entirely, but very similar gameplay, from Superior Software

Conventions and Nomenclature

The guide is written from the viewpoint of programming in C++, although I actually program in Delphi (Pascal). As Repton is mainly from the Acorn world, most code would be shown as BBC BASIC V or ARM assembly.

Keywords

Throughout this guide, I have used some common keywords:
XOR: bitwise eXclusive OR
AND: bitwise AND
OR: bitwise OR
Also, in BBC BASIC sections, EOR is the same as XOR.

Numeric Notation

I have also used C++ conventions to represent hexadecimal notation…i.e. 0x10 is 10 in hexadecimal (which is equivalent to 16 in decimal).
Bits and bytes are counted from 0, with bit 0 being the least significant, or right most bit. Bytes, on the other hand, are counted as you encounter them. So byte 0 would be the left most byte (which is normally the most significant byte).

Endian Notation

With a hex number of 0x1234, 0x12 would be considered the MSB (most significant byte), while 0x34 would be the LSB (least significant byte). This would be represented, and stored in these data files, as LSB/MSB. Therefore, you would encounter them as 0x34, 0x12 – this is known as "Little Endian". If it were MSB/LSB, then this is known as "Big Endian", and will be noted as such.

How The Maps Are Stored

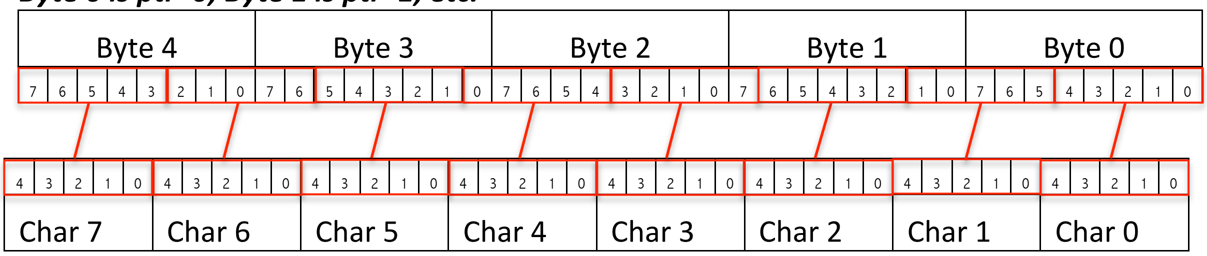
The level data, or maps, are stored as a common format on all platforms covered by this book, except for EGO: Repton 4, which has a different format completely (as it was originally another game entirely, and had it's name and characters changed to be another Repton).

The size of each level changes between levels (Repton and Repton 2 being 32x32, and Repton 3 being 24x28 characters), but the actual storage is the same. Eight characters are packed into five bytes, which basically allows for up to 32 characters on the maps. In reality, there are more characters, but the extras are the animation ones which do not need to be included on a map. This said, there are actually two variations on this method.

Method One

The eight characters are placed on the map as you would read, i.e. from left to right. The bits that make up the character number 0-31 are split thus:

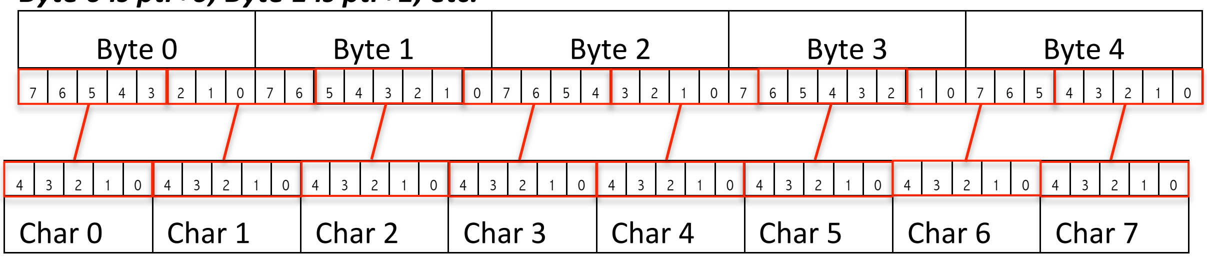
Byte 0 is ptr+0; Byte 1 is ptr+1; etc.



Char 0 is x+0,y; Char 1 is x+1,y; etc. and bits 5-7 of each character are 0.

Method Two

Byte 0 is ptr+0; Byte 1 is ptr+1; etc.



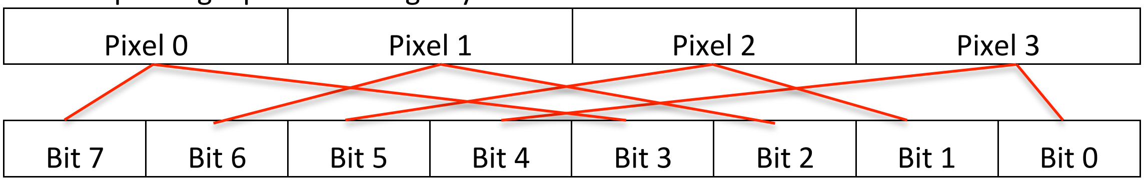
How The Graphics Are Stored

How the graphics are stored will depend on the target machine. With the 8 bit machines, the character sizes are 16x32 pixels (12x24 for Electron Repton 3, and 8x16 for Electron Repton 1, 2 and Infinity). The Archimedes and RISC OS are much simpler, although the character sizes are now 32x32 pixels, as they have more colours to play with. Each 4 bits represent a single pixel, so you can get 2 pixels from a single byte, and each pixel has a possible 16 colours to choose from. The Hi-Res format of Desktop Repton 3 goes even further, not only doubling the size of the characters (to 64x64), but also doubling the bit storage. Each byte represents a single pixel (one of 256 possible colours).

This format of storing maps and characters changed with Repton: The Lost Realms, although the 'retro' graphics for the PC version remained. However, the high-resolution graphics for the PC utilised 64x64px 8bpp Windows Bitmap format (the r3g files being a series of Windows Bitmaps with the headers and palette information stripped off).

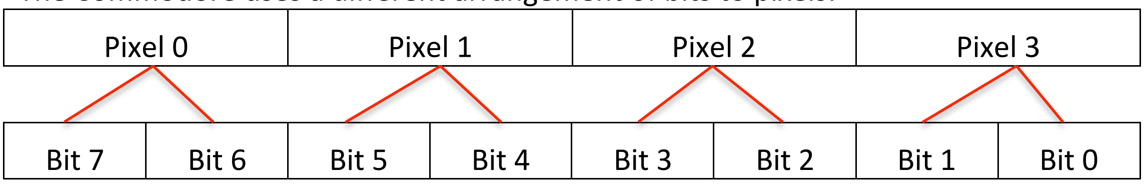
BBC Micro/Acorn Electron

The BBC Micro and Acorn Electron versions of the games all run in MODE 5 and, hence, match the way that the MODE 5 memory is arranged. Each two bits of data make up a single pixel allowing any one of the 4 MODE 5 colours:



Commodore 64

The Commodore uses a different arrangement of bits to pixels:



ZX Spectrum

The ZX Spectrum uses separate data to define the graphics. The pixel data can be either foreground (bit is set) or background (bit is clear). Each byte, in the pixel data, will therefore represent 8 pixels. Then each 8x8px block is described by a single byte:
bit 7 = Flashing
bit 6 = Bright
bits 5,4 & 3 = Paper (background)
bits 2,1 & 0 = Ink (foreground)
The 3 bit colour definitions are described as Green, Red then Blue for bits 2, 1 and 0 respectively. This will make the colours:

0: Black	1: Blue	2: Red	3: Magenta
4: Green	5: Cyan	6: Yellow	7: White

Combined with the Bright flag, this will give a total of 16 colours.

The ZX Spectrum screens have a resolution of 256x192 pixels. The memory is made up thus:

```
15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
0  1  0  Y7 Y6 Y2 Y1 Y0 Y5 Y4 Y3 X4 X3 X2 X1 X0
The 010xxxxxxxxxxxxx is the base address of 0x4000 (start of screen memory)
```

The Y co-ordinates are in pixels down, while the X co-ordinates are in bytes across. The screen memory is from 0x4000 to 0x57FF, with the colour data from 0x5800 to 0x5AFF.

Archimedes/RISC OS

For the two pixels per byte method used with Archimedes Repton 1, 2, and 3; RiscPC Repton 2; & Desktop Repton 1, 2, and 3 (low res), the byte is split in two with bits 0-3 being the first pixel, and 4-7 being the adjacent pixel.

BBC and Electron Repton

Data is spread across the two files REPTON1 and REPTON2 on the BBC. The majority of the data is in the second file, whilst a minimal amount is to be found in the first. All of the data for the Electron versions must be XORed with 0xFF first.

File Format (BBC)

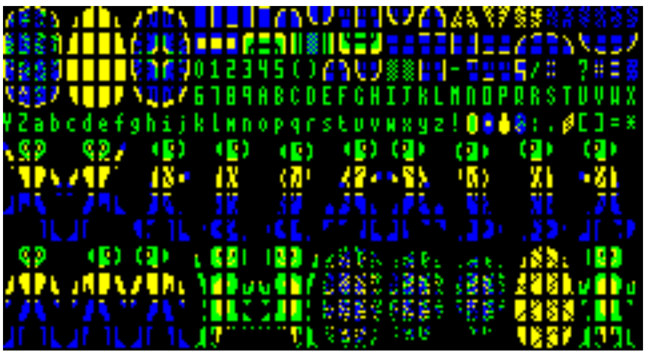
Offset	Description
File: REPTON1	
0x1702	Game playing sprites tile lookup
0x1754	Palette
0x17BF	Small map characters tile lookup
File: REPTON2	
0x1C58	Passwords
0x1EC0	Sprite definitions
0x2F00	Map sprite tile offsets
0x3100	Maps

File Format (Electron)

Offset	Description
0x00E9	Palette
0x00F5	Small map characters tile lookup
0x1B70	Passwords
0x2500	Sprite definitions
0x2AC0	Map sprite tile offsets
0x2C00	Maps

Characters

Sprites are made up of a number of tiles. These tiles, as sprites, are stored as 4x8px tiles, of which there are 520 (224 in Electron) of them:



There is then a lookup table to define which tiles make up the appropriate characters, of which there are 52. As the characters are ultimately 16x32px (8x16px for Electron), this means 16 tiles (4x4), or 4 tiles (2x2) for Electron, make up a single character. The format of the sprites is Method One as described on page 7.

The game playing sprites on the BBC has direct pointers to the top left tile. However, this only has 10 of the 14 sprites, and they are all 4x4. The first location contains the LSB, while the second contains the MSB. You will need to add 0x0F00 for the offset into the file (as the address given is the direct memory address when the file is loaded into memory).

Maps

These are stored, as Method One, described on page 7, to make up the 32x32 character maps, of which there are 12 in total, making each level a total of 640 bytes each. One thing to note is that Repton does not actually appear on the level definitions, as he begins from the same spot on every level (being at 5,5).

Map Characters

These are stored the same as the main characters, with the tile lookup pointing towards where the definition is. This time, the map characters only take up 1 tile each.

Passwords

The passwords are stored unencrypted, and are a maximum of 12 characters long, terminated by Carriage Return (13 or 0x0D).

Palette

The palette is simply 1 byte per screen, with each byte representing a BBC colour (see page 25), which will redefine logical colour 1 (Repton's trousers).

Time Limits

The time limits for all screens in Repton are set at 6000.

Desktop Repton

The data file for Desktop Repton is encoded using data from the sprite file. This file itself has had some encryption done to it to produce the final key, which is then used to decrypt the data.

As this file, encrypted, can be saved and used as a key, it seems sensible to use the code found within the Desktop Repton directory structure to encrypt it once, and saved for future use.

Encryption Key

BBC BASIC program to produce a key to decode the maps:

```
REM Program to process the Desktop Repton 1 Sprite file
REM for use as a key to decode the maps
REM
REM This code is taken directly from the DR1 main program
:
ONERRORREPORT:PRINT" at line ";ERL:END
savepath$="ADFS::RISCOS4.$"
:
REM First we'll extract the code
REM This is stored as code length, code, code function names and pointers
Z%=OPENIN"<Repton$Dir>.Resources.Code"
INPUT#Z%,A%
DIMcode% A%
size=A%
SYS"OS_GBPB",4,Z%,code%,A%TO,,,A% : REM Read bytes from current pointer
:
REM Now we need to load the Sprite file in and process it
SYS"OS_File",17,"<Repton$Dir>.Resources.Sprites" TO,,,,D% : REM Get file length
D%+=48+655360
DIM sprs% D%
!sprs%=D%
SYS"OS_File",16,"<Repton$Dir>.Resources.Sprites",sprs%+4 : REM Load file into sprs%+4
:
REM These are the functions within the code that we'll be using
sprtab=&C74+code%
process=&1590+code%
map=&500+code%
DIM map% 1600
B%=map%
!map=B%
sprtab!-4=sprs%
FORn%=0TO50
 SYS"OS_SpriteOp",256+24,sprs%,STR$n% TO,,A
 A%+=&2C
 sprtab!(n%*4)=A%
 CALLprocess
 IF A%!-8>A%!-12 THEN A%+=512:CALLprocess
NEXT
:
REM Now we save the key to a file
SYS"OS_File",0,savepath$+".DR1Key",,,sprs%+4,sprs%+18004
PRINT"Desktop Repton 1 decode key saved in "savepath$
```

You will then need to use this key to decrypt the file, as shown in the following Pascal code. For the crypt procedure:

```
procedure crypt(var data,enc_data: array of Char;ptr,enc_ptr,amt,seed: Integer);
```

data is the datablock where the file is loaded into.
enc_data is the datablock for the above key.
ptr is the pointer into data that will be decoded.
enc_ptr is the pointer into enc_data that will be used.
amt is the size of data that needs decoding.
seed is the decryption seed.

For the initial decryption, which is done over the entire file, the seed is the last word (four bytes) of the data block ORed with itself shifted left 16 times, or:

```
seed=seed OR (seed<<16)
```

After the entire file has been decrypted, you will then need to decrypt the password and author area.

```
begin
 temp:=Ord(data[size-4])+Ord(data[size-3])shl 8+Ord(data[size-2])shl 16+Ord(data[size-
1])shl 24;
 temp:=temp OR (temp shl 16);
 getResource('DR1Decode',spr_data); {Get the decryption key}
 crypt(data,spr_data,0,12,size-4,temp);
 for screen:=0 to levels-1 do
  crypt(data,data,(((levels*1024)+levels+3)AND-4) +(screen*68),screen*1024,68,29);
end;
procedure crypt(var data,enc_data: array of Char;ptr,enc_ptr,amt,seed: Integer);
var
 c,d: Integer;
begin
 repeat
  c:=Ord(enc_data[enc_ptr])+Ord(enc_data[enc_ptr+1])shl 8+Ord(enc_data[enc_ptr+2])shl
16+Ord(enc_data[enc_ptr+3])shl 24;
  enc_ptr:=enc_ptr+4;
  d:=Ord(data[ptr])+Ord(data[ptr+1])shl 8+Ord(data[ptr+2])shl 16+Ord(data[ptr+3])shl
24;
  c:=seed+(seed*c);
  d:=d+seed;
  d:=d XOR ((c shr 12)+(c shl 20));
  d:=d XOR ((c shr 20)+(c shl 12));
  d:=d-seed;
  data[ptr+3]:=chr((d AND $FF000000)shr 24);
  data[ptr+2]:=chr((d AND $00FF0000)shr 16);
  data[ptr+1]:=chr((d AND $0000FF00)shr 8);
  data[ptr+0]:=chr(d AND $000000FF);
  ptr:=ptr+4;
  amt:=amt-4;
 Until amt=0;
end;
```

File Layout

Size of file is (((levels x 1093) + 7) AND -4)

Offset	**Length**	**Description**
0x0000	0x400*levels	Maps (32x32 chars)
((levels*0x400)+3)AND-4	0x44*levels	Info block (see below), encoded twice
length-4	0x04	Encryption seed
Level Info Block		
0x00	0x20	Password
0x20	0x20	Author
0x40	0x04	Time Limit

Characters

The characters used for Desktop Repton are stored as standard RISC OS sprites in a separate file, *Resources.Sprites*. You will need to refer to the RISC OS Programmer's Reference Manual for details on how these are stored.

Archimedes and RiscPC Repton

File Layout

Offset	Length	Description
0x03A44	0x01080	Map Characters
0x04AC4	0x01E00	Maps
0x068E4	0x1A000	Characters
0x208E4	0x00084	Passwords
0x20968	0x00300	Palette
0x20C74	0x00030	Time Limits

Characters

The characters to display the map are 16 x 16px, with 2px per byte. This means each character is stored in 128 bytes, of which there are 33 of them. The characters used for playing are 64 x 64px, and again, 2px per byte. Each character uses 2048 bytes of space of which there are 52 of them. Each pixel is a colour number, referenced into the palette data.

Maps

These are stored, as Method One, described on page 7, to make up the 32x32 character maps, of which there are 12 in total, making each level a total of 640 bytes each.

Time Limits

These are stored as 4 bytes each: LSB, MSB, 0x00, 0x00. Note, that these are not all 6000, as BBC/Electron Repton.

Passwords

The passwords are stored as 11 bytes per level for each of the 12 levels. They are encoded using the map data:

character: ASCII value of each character
x: offset into the password (0..10)
passwords[]: password data for screen
mapdata[]: map data for screen

```
character = ( passwords[x] XOR mapdata[47 * (x+1)] ) AND 0x1F OR 64
```

then, if character>13:

```
character = character AND 0xDF
```

finally, if character is an '@' then replace for ASCII 13

Palette

The values stored at this location are the red, green and blue components for each of the 16 colours per level. They are stored 0x10, RR, GG, and BB.

BBC and Electron Repton 2

The BBC file D.RepB needs to be XORed with 0x66 to get any useable data. The Electron version of Repton 2 is split into two files on the DFS version, while on the ADFS version it is a single file.

File Layout (BBC)

Offset	Description
0x014A	Sprite sizes
0x0169	Sprite position offsets (LSB)
0x0188	Sprite position offsets (MSB)
0x01C5	Palette
0x0FD2	Start Position (x)
0x0FD6	Start Position (y)
0x1B00	Level Data offsets
0x1B40	Transporters
0x1CF8	Puzzle Piece Definitions
0x1DA0	Sprite Tile offsets
0x2240	Sprite and Puzzle Piece Definitions
0x2680	Palette
0x3500	Level Definitions

File Layout (Electron, DFS format)

File	Offset	Description
ReptonA	0x187A	Sprite sizes
ReptonA	0x1899	Sprite position offsets (LSB)
ReptonA	0x1939	Sprite position offsets (LSB)
ReptonA	0x1958	Sprite position offsets (MSB)
ReptonA	0x1AB3	Palette
ReptonB	0x0B60	Start Position (x)
ReptonB	0x0B64	Start Position (y)
ReptonB	0x1600	Sprite and Puzzle Piece Definitions
ReptonB	0x1720	Puzzle Piece Sprites
ReptonB	0x18A0	Puzzle Piece Definitions
ReptonB	0x1950	Transporters
ReptonB	0x1B00	Level Data offsets
ReptonB	0x1B40	Text characters
ReptonB	0x1E40	Sprite Tile offsets
ReptonB	0x24C0	Palette
ReptonB	0x2900	Level Definitions

Level Data

The offset data is 4 bytes per level, with each byte being an offset into the level definitions: *addr+(offset x 160)*. Each level is 32x32, and each offset representing 8 rows of a map.

Character numbers are shared on three occasions – the Finish character only appears on Screen A, while on the other screens it is a spirit; and transporters and puzzle pieces do not appear on the level data, so you will need to fall back on the respective data (see below).

Any offset byte of 0x80 or more is eight rows of repeating character, the least significant four bits indicating the character number (0=blank, 6=diamond, etc.)

The level definitions are uses Method One, as described on page 7.

Puzzle Piece Definitions

These are stored in puzzle piece order (as put together using the sprite data below), and each one is stored as four bytes: screen, x, y, and location on screen A. You will need to AND the screen byte with 0x0F. There are 42 puzzle pieces. The final byte, location on screen A, is a pair of co-ordinates with the x being bits 0-3, and the y being bits 4-7. This is an offset from position 10,24.

Transporters

These are stored as six bytes: source screen, x, and y (i.e. where the transporter is) and destination screen, x, y (i.e. where it takes Repton to). There are 64 transporters.

Start Position

Where Repton starts the game, on Screen A (as x, y co-ordinates) – should be 16,7.

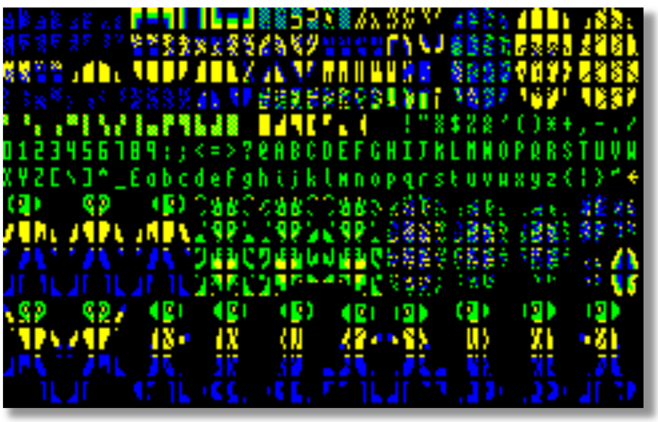
Palette

The palette is stored in two locations. The first location is four bytes. Each byte is made up of the logical colour (top four bits), and the actual colour (lower four bits). To get the appropriate palette for the level, you will need to AND the level number with 3 (with the level number being 0..15) to produce an offset of 0, 1, 2 or 3.

However, this data is only valid for levels 2..15. For 0 and 1, there is a second location with two bytes. Also, logical colour 0 is always actual colour 0 (black), and logical colour 3 is always actual colour 2 (green). And finally, level P (15) appears to be a combination of offset 2 and 3 of the first location.

Game Sprites and Puzzle Piece Sprites

As Repton before it, these are stored in tiled format. The sprite tiles are 4x8px tiles of which there are 600 of them. These make up the final sprites by taking the sprite definition data, where each byte is an offset into these tiles.



These are for the map and puzzle piece sprites only. The playing sprites are stored differently. For these, there are a number of bytes that determine how big each one is – there are 30 on the BBC and 31 on the Electron (although only 25 are used). These can be 1, 2, 3 or 4 (for 1x1, 2x2, 3x3 or 4x4). Following on from this are the addresses of the offsets for the top left tile. This offset will be the direct address after the file is loaded into memory. To get the offset into the file, just subtract 0x0D00 for the BBC and 0x1100 for the Electron.

However, the Electron is not so as simple as that, as there are two locations for the LSB address. The guide seems to be:

Valid for when x is from 0 to 3, and 9 to 30. What 4 to 8 are used for is unclear at the time of writing.

1. Read the sprite size from location 0x187A + x;
2. Read the MSB address from location 0x1958 + x;
3. When x <= 22, read the LSB address from location 0x1899 + x; and
4. When x >= 23, read the LSB address from location 0x1939 + x.

Desktop Repton 2

Encryption Key

As Desktop Repton 1, the data is encoded with the processed sprite file. The encoding, and processing, of Repton 2 is almost identical to Repton 1. Therefore, the following is an almost identical BBC BASIC program to extract and process the Sprite file for use as a key to decode the Maps data file:

```
REM Program to process the Desktop Repton 2 Sprite file
REM for use as a key to decode the map
REM
REM This code is taken directly from the DR2 main program
:
ONERRORREPORT:PRINT" at line ";ERL:END
savepath$="ADFS::RISCOS4.$"
:
REM First we'll extract the code
REM This is stored as code length, code, code function names and pointers
:
DIMmap% 1920,map1% 1920
Z%=OPENIN"<Repton2$Dir>.Resources.Code"
INPUT#Z%,A%
DIMcode% A%
size=A%
SYS"OS_GBPB",4,Z%,code%,A% TO,,,A% : REM Read bytes from current pointer
:
REM Now we need to load the Sprite file in and process it
SYS"OS_File",5,"<Repton2$Dir>.Sprites" TO,,A%,,D%
IF (A%>>8)=&FFFFFFCA THEN
 B%=OPENIN"<Repton2$Dir>.Sprites"
 SYS"OS_GBPB",4,B%,&860C,20
 A%=EXT#B%-20
 SYS"Squash_Decompress",8,D% TO wksz%
 D%+=41136
 DIMsprs% D%
 !sprs%=D%
 C%=END+1024
 SYS"OS_GBPB",4,B%,C%,A%
 CLOSE#B%
 B%=-1
 SYS"Squash_Decompress",,code%,C%,A%,sprs%+4,D%
ELSE
 D%+=41136
 DIMsprs% D%
 !sprs%=D%
 SYS"OS_File",255,"<Repton2$Dir>.Sprites",sprs%+4
ENDIF
:
sprtab=&9F4+code%
process=&13CC+code%
map=&DDC+code%
sprtab!-4=sprs%
B%=map%
!map=B%
FOR n%=0 TO 54
 SYS"OS_SpriteOp",256+24,sprs%,STR$n% TO,,A%
 A%+=&2C
 sprtab!(n%*4)=A%
 CALLprocess
 IF A%!-8>A%!-12 THEN A%+=512:CALLprocess
NEXT
:
REM Now we save the key to a file
SYS"OS_File",0,savepath$+".DR2Key",,,sprs%+4,sprs%+26316
PRINT"Desktop Repton 2 decode key saved in "savepath$
```

And, as Desktop Repton 1 , the file will need to be decrypted twice – once overall, then once each for each level:

```
temp:=Ord(data[size-4])+Ord(data[size-3])shl 8+Ord(data[size-2])shl 16+Ord(data[size-
1])shl 24;
temp:=temp OR (temp shl 16);
```

```
crypt(data,spr_data,0,12,size-4,temp);
for screen:=0 to levelinfo[1]-1 do
 crypt(data,data,(((levelinfo[1]*1024)+levelinfo[1]+3)AND-4)
+(screen*68),screen*1024,68,29);
```

File Format

Offset	Length	Description
0x0000	0x6000	Maps
0x6000	0x0048	Offsets to level data
0x6084	0x0011	Palette offsets
0x60A4	0x0011	Edge
0x63C4	0x00FC	Puzzle piece definitions
0x6544	0x0180	Transporter data

Co-ordinates

The x and y co-ordinates are not screen co-ordinates, but refer to a direct memory location into the level data. However, this level data has been altered to include 4 rows of edge walls on either side. To convert from these
x,y to actual level x,y positions use:

```
screen_x = (( y * 10) + (x DIV 4) - 164) DIV 40
screen_y = (( y * 10) + (x DIV 4) - 164) MOD 40
```

Maps

For each of the 17 levels, there are 4 bytes which determine the offset into the file where the map data is held (LSB/MSB/0x00/0x00). However, there are 18 entries – the offset to the next level is used to determine level size.

Edges

This determines whether the edge is a wall (0x18) or a barrier (0x19). However, this can be any character, as the value stored here is just the character number.

Palette

This is 1 byte per level, which is an indication to which of the 48 bytes in the RepPal file that the colour data is held for this level (3 bytes per colour, of which there are 16).

Puzzle Pieces

These are 6 bytes for each of the 42 pieces. They are stored as two sets of data: where found, where placed. Each set is level, x, y. Normally, where the piece will be placed will be on level A, at the bottom.

Transporters

Simply, these are 6 bytes for each of the 64 transporters. As the puzzle pieces, they are in 2 sets: from, to. Each set is then, as the puzzle pieces, level, x, y.

Characters

The characters used for Desktop Repton are stored as standard RISC OS sprites in a separate file, *Sprites*. You will need to refer to the RISC OS Programmer’s Reference Manual for details on how these are stored. The puzzle pieces are stored as a single sprite called ‘puz’.

This reports itself to RISC OS as a Mode 0 sprite with a width of 64 and a height of 42. However, if you look at the data from offset 0xF4E8 into the file, you can see that it is a series of bits. Each 8 bytes will describe a single puzzle piece - 8 bytes x 8 bits = 64 bits or 64 pixels, with 42 pixel height = a row for each piece.

So, a value of 0xFE is 11111110 in binary that translates to 7 solid pixels and an empty pixel. As all the other tiles are 32x32px, each bit is used four times across, and four times down (4x8=32). The colouring is applied afterwards - generally, Desktop Repton colour the solid pixels alternate yellow and green (colours 2 and 8 from the RepPal file).

Archimedes and RiscPC Repton 2

File Format (Archimedes)

Offset	Length	Description
0x04D88	0x2800	Maps
0x07588	0xD400	Characters
0x14988	0x0200	Transporter locations
0x14B88	0x00A8	Puzzle Piece data
0x14C30	0x0400	Level colour palettes
0x15030	0x0010	Edges

File Format (RiscPC)

Offset	Length	Description
0x00FA0	0x02800	Maps
0x037A0	0x35000	Characters
0x387A0	0x00200	Transporter locations
0x389A0	0x000A8	Puzzle Piece data
0x38A48	0x00400	Level colour palettes
0x38E48	0x00010	Edges

Maps

These are stored packed, as Method One on page 7. This makes each of the 16 levels 640 bytes each.

Edges

These are ignored in the RiscPC version, but dictate what character surrounds the level. This is quite simply the character number.

Characters

On the RiscPC, these are 64x64px with each byte being 2px, to represent one of the 16 colours. The Archimedes characters are a quarter the size, at 32x32px. There are 106 characters on both versions.

Transporters

The Transporters are stored as 8 bytes each, split into two sets – from and to. Each set is screen,x,y,0x00. The co-ordinates are worked out thus:

```
y = ((byte1 + (byte2 * 256)) - 324) DIV 40
x = ((byte1 + (byte2 * 256)) - 324) - (40 * y)
```

Puzzle Pieces

The four bytes are screen, x, y, Screen A position. Similar to the transporters, the co-ordinates will need to be converted:

```
y = ((byte1 + ((byte2 AND $F) * 256)) - 324) DIV 40
x = ((byte1 + ((byte2 AND $F) * 256)) - 324) - (40 * y)
```

The Screen A position co-ordinates are worked out:

```
y = ((((byte2 AND $F0) >> 4) + (byte3 * 16)) - 84) DIV 40
x = ((((byte2 AND $F0) >> 4) + (byte3 * 16)) - 84) - (40 * y)
```

Palette

For each level, there are 16 colours and each are stored in their Red, Green and Blue component form: 0x10, RR, GG, BB.

Repton 3

The format for the Electron, BBC, Archimedes and Desktop Repton versions are largely similar. The Commodore 64 version differs slightly. The one main difference is there are extra bytes, seemingly unused, in the Commodore file:

0x0000: 0xA0 *0x0001: 0x67* *0x2752: 16 bytes of 0x00*

Also, each format of file has a specific size. This will be:

BBC Micro: 9,760 bytes (0x2620)
Acorn Electron: 7,712 bytes (0x1E20)
Commodore 64: 10,210 bytes (0x27E2)
Acorn Archimedes and Desktop Repton Low Res: 28,832 bytes (0x70A0)
Desktop Repton High Res: 102,560 bytes (0x0190A0)

File Layout (BBC/Electron)

Offset	Length	Use
0x0000	0x40	Passwords
0x0040	0x10	Time Limits
0x0050	0x10	Edit Codes
0x0060	0x80	Transporters
0x00E0	0x20	Colour Palette
0x0100	0xD20	Maps
0x0E20	0x1800	Characters (BBC)
0x0E20	0xD80	Characters (Electron)

File Layout (Archimedes/Desktop Repton)

Offset	Length	Use
0000	0x40	Passwords
0040	0x20	Time Limits
0060	0x10	Edit Codes
0080	0x100	Transporters
0180	0x200	Colour Palette
0380	0xD20	Maps
10A0	0x6000	Characters

File Layout (Commodore)

Offset	Length	Use
0x0000	0x02	Unused (should be 0xA0 and 0x67)
0x0002	0x40	Edit Codes
0x0042	0x20	Colour Palette
0x0062	0x1800	Characters
0x1862	0x180	Map Characters
0x19E2	0x10	Time Limits
0x19F2	0x40	Passwords
0x1A32	0xD20	Maps
0x2752	0x10	Unused (should all be 0x00)
0x2762	0x80	Transporters

Passwords

On the Commodore 64, these are not encoded in any way, and are padded with Spaces (0x20). The other versions are terminated by a Carriage Return (0x0D) and are not padded – the extra bytes are ignored.

For the BBC and Electron, each byte is XORed with 63-offset:
Byte 0 XOR 63 Byte 1 XOR 62 to Byte 63 XOR 0
thereby covering all eight passwords, each one being up to eight characters long.

The Archimedes and Desktop Repton versions use a formula to encode/decode the data:

Archimedes/DR Low Res:
```
char XOR (mapdata[(charoffset+1) * 0x2F] AND 0x1F
```
DR High Res:
```
char XOR (mapdata[((charoffset+1) * 0x2E)-3] AND 0x1F
```

Edit Codes

The BBC and Electron is simply 2 bytes per Edit Code, arranged as LSB/MSB. The Commodore stores them as a string, padded by zeros at the beginning (to 5 characters). However, as the user cannot enter an edit code in the Commodore version of the game, the edit codes are five bytes of ASCII 0x00 per screen with user-defined scenarios.

The Archimedes and Desktop Repton versions, although are stored within the file (which are a red herring), are actually worked out from the map data:

"If the raw map data byte has bit 2 set then add double the raw map data byte, otherwise just add the raw map data byte." Or, in BBC BASIC:
```
REM screen is the screen number (0-7)
REM and data is a memory location where the data file is loaded
code=0
FOR x=0 to 419
 ptr=&380+(screen*420)+x
 IF (data?ptr AND 4) THEN code=code+(data?ptr)*2 ELSE code=code+data?ptr
NEXT
```

Time Limits

The BBC and Electron versions are 2 bytes per screen, while the Archimedes and Desktop Repton versions are 4 bytes per screen, but with only the first two bytes used (the last two should be 0x00). These 2 bytes are the time limit stored as LSB/MSB.

The Commodore stores them as Binary Coded Decimal…i.e., 0x12 and 0x34 would be a time limit of 1234.

Transporters

The 4 transporters per screen are stored in four bytes each on the BBC, Electron and Commodore as source x, source y, destination x, destination y. If unused, the first byte should be 0xFF.

The Archimedes and Desktop Repton versions are stored as 8 bytes per transporter, with bytes 0 and 1 as the source, and bytes 4 and 5 as the destination. Bytes 2, 3, 6, and 7 should be 0x00. The x and y are worked out thus:
```
y = ((byte0 + byte1 * 256) - 148) DIV 36
```

```
x = ((byte0 + byte1 * 256) - 148) - (36 * y)
```

It would be worth validating this data once you have unpacked the map data to ensure that there is actually a transporter at the source location.

Colour Palette

Each screen can be a different colour, so each of the four colours are stored in this location. For the BBC and Electron, each of the 4 colours per screen are stored as a single byte per colour (meaning 4 bytes per screen). Each byte represents a BBC actual colour:

0: Black 1: Red 2: Green 3: Yellow 4: Blue 5: Magenta 6: Cyan 7: White
(or, bit 0: red, bit 1: green, and bit 2: blue components i.e. 2 bits per pixel)

The Commodore is similar, but the colours are different:

0: Black	*1: White*	*2: Red*	*3: Cyan*	*4: Purple*
5: Green	*6: Blue*	*7: Yellow*	*8: Orange*	*9: Light Orange*
10: Light Red	*11: Light Cyan*	*12: Light Purple*	*13: Light Green*	*14: Light Blue*
15: Light Yellow				

The Archimedes and Desktop Repton stores them as 4 bytes per colour, 16 colours per screen. Byte 0 is unused, while the first 4 bits of bytes 1, 2 and 3 are the red, green and blue components of the colour, i.e. 4 bits per pixel. The High Res version of Desktop Repton goes that step further with all eight bits being used, making it 8 bits per pixel.

Maps

The map data across all four platforms is exactly the same, and is encoded as Method One described on page 7. The Repton 3 maps are 28 characters across by 24 characters down, which means that each screen takes up 420 bytes each.

Characters

The character data is stored as described on page 8. However, they are different to previous Reptons in that they are not stored as tiles, but the entire character can be found in a single 16x32 pixel definition, per character (12x24 for Electron, 32x32 for Archimedes and Desktop Low Res, and 64x64 for Desktop High Res).

Competition Codes

The competition code, on the BBC and Electron, is worked out from the maximum possible score, and a hash table calculated from the first 255 bytes of the file.

For the maximum possible score, diamonds (and, hence, safes and cages) are scored at 5 points, eggs are 20, and crowns are 50. The following C++ function (where buffer is the area of memory the file is loaded into, and score is the aggregate maximum possible score of all 8 screens):

```
void calculate_competition_code(unsigned char *buffer, unsigned int score)
{
  unsigned char codelen;
  unsigned char comp[20];
  uint32_t roller;
  uint16_t hash;
  uint16_t loop;
  unsigned char output;
  // Start with the maximum score for this level set
  roller=score;
  // Seed the hash
  hash=0xeeff;
```

```
  // Calculate "HASH" of first 255 bytes of level file
  //   PASSWORDS/TIME_LIMITS/EDIT_CODES/TRANSPORTERS/COLOUR_PALETTE
  for (loop=0; loop<=0xff; loop++)
  {
    // Read byte from level file
    output=buffer[loop];
    // Update HASH LOW byte
    output^=(hash&0x00ff);
    hash=(hash&0xff00)|output;
    // Update HASH HIGH byte
    output^=((hash&0xff00)>>8);
    hash=(hash&0x00ff)|(output<<8);
  }
  // Add the hash to the roller
  roller|=(hash<<16);
  codelen=0x00;
  while (roller != 0x00)
  {
    output=0x00;
    for (loop=0x00; loop<0x20; loop++)
    {
      // Shift output by 1 bit
      output=output<<1;
      // If roller MSB then add to output
      if (roller&0x80000000)
        output++;
      // Shift roller by 1 bit
      roller=roller<<1;
      // If output overflows 0..9 then add bit to roller LSB
      if (output>=0x0a)
      {
        output-=0x0a;
        roller++;
      }
    }
    // Add output number to storage stack
    if (codelen<sizeof(comp))
      comp[codelen++]=output;
  }
  // Read numbers off stack to output competition code
  do
  {
    codelen--;
    printf("%d", comp[codelen]);
  } while (codelen>0);
  printf("\n");
}
```

Repton Infinity

The Repton Infinity files are separated into directories on disc, depending on their contents. The files are usually referred to by their filename including this directory:

E Thumbnail sprites (for the editors)
G Linked game file (i.e. Run this to play)
M Maps
O Compiled object code from the source
S Full size sprites (for the game)
T Tokenised source code
D Combined file for source (Master only)

The Acorn Electron version will have an 'e' before this letter (i.e. eE, eG, eM, etc.). All Electron files are identical to the BBC files, except for those indicated below.

G.game format

The whole file is encrypted with a basic EOR scheme of:

```
byte EOR key
```

The key starts at 0 and is decreased by 3 each cycle, so we have:

```
byte0 EOR 0
byte1 EOR 0xFD
byte2 EOR 0xFA
```

As 'junk' bytes are put in to fill in space, the file should always be the same size:

0x0000 Sprites (as 0x114 from S.*)
0x1800 *[junk byte]* Map Author Name (as 0x000 from M.*)
0x1810 *[junk byte]* Sprite Author Name (as 0x004 from S.*)
0x1820 *[junk byte]* Code Author Name (as 0x000 from O.*)
0x1830 *[junk byte]* Game Title (as supplied by Linker)
0x1850 *[junk byte]* Ending Message (as supplied by Linker)
0x1870 *[junk byte]* Filename (e.g. Rep3A)
0x1880 Object Code (as 0x0F0 from O.*) padded with NULL
0x1B50 Map data (as 0x010 from M.*)
0x2350 Map Sprites (as 0x014 from S.*)
0x2450 Object Code look up table (from 0x010 in O.*)

eG.game format

As G.* format, except:

0x0000 Sprites (as 0x114 from eS.*)
0x0600 *[junk byte]* Map Author Name (as 0x000 from eM.*)
0x0610 *[junk byte]* Sprite Author Name (as 0x004 from eS.*)
0x0620 *[junk byte]* Code Author Name (as 0x000 from eO.*)
0x0630 *[junk byte]* Game Title (as supplied by Linker)
0x0650 *[junk byte]* Ending Message (as supplied by Linker)
0x0670 *[junk byte]* Filename (e.g. Rep3A)
0x0680 Object Code (as 0x0F0 from eO.*) padded with NULL
0x0950 Map data (as 0x010 from eM.*)
0x1150 Map Sprites (as 0x014 from eS.*)
0x1250 Object Code look up table (from 0x010 in eO.*)

E.game file format

48 blocks of 16 bytes, each is a Mode 1 screen dump of thumbnail (at 2bpp = 4ppB) (thumbnail is 8 x 8 pixels), as described on page 8.

M.game format

Header

Offset	Length	Use
0x000	0x010	Name of author terminated with a 0x0D; the rest of the line is junk.
0x010	0x200	Map screen 1
0x210	0x200	Map screen 2
0x410	0x200	Map screen 3
0x610	0x200	Map screen 4

Screen formats

Offset	Length	Use
0x000	0x1E0	Map data
0x1E0	0x018	Teleporters
0x1F8	0x001	Map visible flag (01 = map visible)
0x1F9	0x001	Password flag (01 = requires a password)
0x1FA	0x002	Score - a 16 bit number in LSB/MSB form
0x1FC	0x004	Palette: 1 byte for each colour

Repton Infinity maps have dimensions of 32 x 24 characters with 32 possible characters that can be used. The format used is Method Two as described on page 7.

Transporter data is given as 2 16 bit numbers (in LSB/MSB form) for each transporter: source and destination. Each 16 bit number is an address:

```
X=addr MOD 32
Y=addr DIV 32
```

S.game file format

Header

Offset	Length	Use
0x000	0x0001	Colour 0
0x001	0x0001	Colour 1
0x002	0x0001	Colour 2
0x003	0x0001	Colour 3
0x004	0x0010	Name of author terminated with a 0x0D.
0x014	0x0100	Map sprite chunk
0x114	0x1800	Sprite chunk

Map sprite chunks

Size of image is 4 x 8, and as per format on page 8.

Sprite chunk

Each entry is 128 bytes in size, as per format on page 8. Size of each sprite is 16 x 32 pixels.

eS.game file format

This is identical to the S.* format, except:

Offset	Length	Use
0x000	0x001	Colour 0
0x001	0x001	Colour 1

0x002	0x001	Colour 2
0x003	0x001	Colour 3
0x004	0x010	Name of author terminated with a 0x0D
0x014	0x100	Map sprite chunk
0x114	0x600	Sprite chunk

Sprite chunk

Each entry is 32 bytes in size, as per format on page 8. Size of each sprite is 8 x 16 pixels.

O.game file format

Offset	Length	Use
0x00	0x10	Author Name 0xd terminated
0x10	0x20	Low byte of the address of the DEFINE ACTION routine/sprite
0x30	0x20	High byte of the address of the DEFINE ACTION routine/sprite
0x50	0x20	Low byte of the address of the DEFINE HITS routine/sprite
0x70	0x20	High byte of the address of the DEFINE HITS routine/sprite
0x90	0x20	System flags (1)/sprite
0xB0	0x20	System flags (2)/sprite
0xD0	0x20	User flags/sprite
0xF0	...	6502 machine code for the DEFINE routines, base is 0x5BB0

Addresses are absolute addresses for the routine which is place in memory at 0x5BB0 (so it steals the first few lines of screen memory). If the address is 0; then the routine is not defined.

System Flags (1) are:

0x01: unknown	0x02: unknown	0x04: One	0x08: Two
0x10: unknown	0x20: unknown	0x40: unknown	0x80: Animate

If 0x04 and 0x08 and 0, then speed is Four. It is unknown what happens if both are 1.

System Flags (2) are:

0x01: Transport	0x02: Squash	0x04: Cycle	0x08: Under
0x10: VPush	0x20: HPush	0x40: Deadly	0x80: Solid

Code is translated from the Reptol directly to 6502 code. Then optimised to replace JSR xxxx:RTS with JMP xxxx. This isn't perfect if the code is followed by an END, which will still stay as RTS.

Code equivalents:

(Note all JSRs can be turned into JMPs by the optimising if they are the last statement in the logical flow.)

CHANCE(x): (75% used here)

```
LDA #&FF:STA &33:LDA #&5F:STA &34:JSR &1D43:BCS xxxx (IF)
```

0x5FFF = 75? Some form of floating point. Range is 0 - 99.99

CHANGE(x,y):

```
LDX #x:LDY #y:JSR &1982
```

Where x and y are the sprite numbers to change from (x) to (y)

CONTENTS x:

```
CMP #x
```

Where x is the sprite number (this is returned from the routine for LOOK(x).

CREATE(x,d):

```
LDY #d:LDA #x:JSR &19EB
```

or, if in the HITS section:

```
LDY #d:LDA #x:JSR &1A54
```

Where x is the sprite number

d is the direction, listed below. If this is not included d is 0x42

d is the location to create. 0x42 is the current location on a grid of 32 characters. So W is -1; E is +1; N is -0x20; S is +0x20; NW is -0x21; NE is +0x19; SW is +0x19; NE is +0x21. (Why 0x42?)

DEFINE:

Is removed - reproduce these from the header

EASTOF:

```
JSR &18A5:BCC xxxx (IF)
```

EFFECT(x):

```
LDA #x:JSR &1C13
```

ELSE:

Dealt with by IF

END:

```
RTS
```

Not removed in Optimisation(!) even if preceded by a JSR; so you'll see the strange statement of JMP xxxx:RTS

ENDIF:

Dealt with by IF

EVENT(x):

```
LDA #&0e:AND #x:BNE xxxx (IF)
```

The event timer byte (0x000E) has a bit set for each timer that has gone off, so the AND is for the check of the bit:

6	0x7F
5	0x3F

FLASH(x):

```
LDA #x:STA &0069
```

Where x is the numeric equivalent of the colour, e.g. 3 = yellow

FLIP:

```
LDA (&0058),Y:EOR #&40:STA (&0058),Y
```

0x0058 contains the status byte for the current object, bit 0x40 is the STATE() bit.

GOTO (x):

```
JMP x
```

x will be the absolute address for the label code; i.e. you'll only have a JMP to 0x5BB0 for a GOTO.

HITBY(x):

```
LDA #&0049:BNE xxxx (IF)
```

IF:

Usually just implemented as a branch - this does depend on what's been tested. ELSE will be implemented by a JMP at the end of the first if, so something like:

```
LDA (&0058),Y:AND #&40:BNE .else
...
JMP .next
.else JSR &1864
```

```
...
.next RTS
```

If a JSR has been turned into a JMP by the optimiser, assume that this is the end of the IF branch.

KEY:
```
JSR &18CB
```

KILLREPTON:
```
INC &0010
```

LABEL:

Not assembled, will have to reform it programatically

LOOK(x):
```
JSR x
```
These are implemented as separate routines for each direction:

F: 0x17EA	SW: 0x1836	W: 0x1844
R: 0x17EE	SE: 0x1834	E: 0x1850
B: 0x17F2	NW: ?	S: 0x185A
L: 0x17F6	NE: ?	N: 0x1864

MOVE(x):
```
LDA #x:JSR &19AE
```
Where x is the numeric equivalent of the direction:

0: E	1: S	2: W	3: N
4: F	5: R	6: B	7: L

MOVING:
```
LDA (&0058),Y:BPL xxxx
```
Bit 7 of the object status is the moving flag.

NAME:
Not assembled, with have to reform it programmatically.

NORTHOF:
```
JSR &18B3:BCC xxxx (IF)
```

NOT:
Swaps the logic over in the IF, e.g. BMI instead of BPL.

SCORE(x):
```
LDA #x:JSR &1B37
```

SOUND(x):
```
LDA #x:JSR &1C1D
```

SOUTHOF:
```
JSR &188D:BCC xxxx (IF)
```

STATE(x):
```
LDA (&0058),Y:AND #&40:BNE xxxx (IF)
```

WESTOF:
```
JSR &189B:BCC xxxx (IF)
```
User flags are stored in 0x64

T.game file format

Offset	Length	Use
0	16	Name of author terminated with a 0x0D

Language chunks x 48

Each language chunk is the tokenised source code of the sprite, using the below tokens:

80: NAME	81: HITBY	82: LOOK(	83: DEFINE	84: CREATE(
85: IF	86: MOVING	87: ELSE	88: ENDIF	89: GOTO
8A: NOT	8B: KILLREPTON		8C: CHANGE(	8D: END
8E: SCORE(	8F: SOUND(	90: FLIP	91: EFFECT(	92: FLASH(
93: CHANCE(	94: KEY	95: One	96: Two	97: Four
98: TYPE	99: ACTION	9A: HITS	9B: MOVE(	9C: STATE(
9D: LABEL	9E: EVENT(	9F: CONTENTS	A0: Animate	A1: RED
A2: GREEN	A3: YELLOW	A4: BLUE	A5: MAGENTA	A6: CYAN
A7: WHITE	A8: WESTOF	A9: SOUTHOF	AA: EASTOF	AB: NORTHOF

An 0x0D is counted as a line feed.

Indents are performed by a byte greater than 0xC8, the indent is byte - 0xC8 spaces. So 0xCF will indent 6 spaces.

Each sprite entry is terminated with a 0xFE byte. An empty sprite will just contain 0x0D and the 0xFE terminator. All sprites will have a definition, including the animation sprites (even though they cannot have source code assigned to them).

There may be junk after the 48 entries, this must be ignored.

Passwords

Passwords are encoded using the map data, so the following BBC BASIC program will unencode them for you:

```
DIM map &810,bytes 8
m%=map+&10
:
PRINT"Repton Infinity Password Printer"
INPUT'"Filename M."n$'
OSCLI"LOAD M."+n$+" "+STR$~(map)
FOR level=0 TO 2
 PRINT CHR$(level+50)") "FNcalcpass(level)
NEXT level
END
:
DEFFNcalcpass(level)
LOCAL A,X,Y,pword$
FOR Y=0 TO 8
 bytes?Y=0
NEXT Y
Y=0
REPEAT
 FOR X=7 TO 0 STEP-1
  A=bytes?X
  A=A EOR ?((level*&200)+m%+Y)
  A=A EOR ?((level*&200)+m%+&E0+Y)
  bytes?X=A
  Y=(Y-1)AND255
 NEXT X
UNTIL Y=0
FOR X=7 TO 0 STEP-1
 A=bytes?X
 A=A MOD 26
 A=A+65
 bytes?X=A
NEXT X
pword$=""
FOR Y=6 TO 0 STEP-1
 pword$=pword$+CHR$(bytes?Y)
NEXT Y
=pword$+CHR$(bytes?7)
```

Sinclair ZX Spectrum Repton Mania

This is based on the DSK file of an original +3 disc copy of Repton Mania. The BIN file is the DSK file with the 'Track-Info' stripped out (I wrote a small routine to do this and produce the BIN file). The beginnings of tracks are identified by the string "Track-Info", which indicates a 0x100 area of track description that can safely be removed for the purposes of data extraction. The initial 0x1300 bytes can also be removed, but as this does not "get in the way" of any data, it is irrelevant whether it stays or not.

The memory offset column is where the data resides within the actual ZX Spectrum when loaded. From this, the respective positions can be located within a snapshot file (e.g. z80). Remember that 0x4000-0x57FF and 0x5800-0x5AFF are the screen memory locations for pixel data and then colour data respectively - see the notes below the breakdown.

DSK and Memory Usage

.bin Offset	Memory Offset	Description
0x00000		Disc descriptor and disc protection loading code
0x01100		Data - loading code and game selector?
0x01300		Repton Mania Loading screen (0x1800 size)
0x02B00		Repton Mania Loading screen colour data (0x300 size)
REPTON 1		
0x02E00		Repton Loading screen (0x1800 size)
0x04600		Repton Loading screen colour data (0x300 size)
0x04900	0x5B00	Repton Maps (12 off at 0x400 size each)
0x07900	0x8B00	Character set graphics (each character is 5 bytes = 8x5px)
0x07A40	0x8C40	"Repton" display at top of opening screen
0x08240	0x9440	unknown
0x08442	0x9642	Lookup table into the screen memory
0x08772	0x9972	Repton 1 map characters (each character is 8 bytes=8x8px: 32off)
0x08872	0x9A72	Repton 1 character area (each character is 0x80=32x32px: 54off)
0x0A372	0xB572	Character colour data (0x10 per character)
	0xB8D2	End of colour data + blanked off work area
0x0A6F2		unknown
0x0A7B6		Lookup table into the screen memory
0x0A826	0xD60A	Passwords - 10 characters, padded with spaces (ASCII 32)
0x0A89E	0xD682	unknown
0x0A8AA	0xD68E	Level palette colours (5 bytes per level)
0x0A8E6	0xD6CA	unknown
0x0AB82	0xD967	Text labels
0x0AFCF	0xDDB2	Null data
0x0B0C4		unknown - code?
0x0B70C		Null data
0x0B894		unknown - code?
0x0C34D		Null data
0x0C44C		unknown - patch?

0x0CCB6		Null data
REPTON 2		
0x0CD00		Repton 2 loading screen (0x1800 size)
0x0E500		Repton 2 loading screen colour data (0x300 size)
0x0E800	0x5B00	Repton 2 maps, each character is packed into 5 bits
0x10920	0x7C20	Transporter data: 7 bytes (screen from,X,Y,screen to,X,Y,0x00)
0x10AE0	0x7DE0	Puzzle Piece data: 6 bytes (screen,X,Y,ScreenA X,ScreenA Y,0x00)
0x10BDC	0x7EDC	Spirit Data: 4 bytes (Screen,X,Y,direction)
0x10D08	0x8008	"Message to fill 36 spare bytes GJS"
0x10D2C	0x802C	Character set graphics (each character is 5 bytes = 8x5px)
0x10E6C	0x816C	"Repton 2" display at top of opening screen
0x1166C	0x896C	Lookup table into the screen memory
0x1199C	0x8C9C	Music data
0x11A9E	0x8D9E	Repton 2 puzzle pieces (8x8px, each piece is 0x08)
0x11BEE	0x8EFE	Repton 2 character area (32x32px, each character is 0x80)
0x1396E	0xAC6E	Repton 2 character colour data (0x10 per character)
	0xB068	Work area
0x13E00	0xD0FC	Level colour palette (5 bytes per level)
0x13E5A	0xD156	Level size in rows (1 byte per level)
0x13E6C	0xD168	Level offsets (2 bytes per level, add 0xE800 to get offset into file)
0x13E90	0xD18C	Level surrounds (3 bytes per level: top, sides, bottom)
0x13EC6	0xD1C2	Top of level (01 - Meteors, 00 - no meteors)
0x13ED8	0xD1D4	Sequence of colours to animate Repton 2 banner
0x13F19	0xD215	Keyboard mapping
0x13F68	0xD264	unknown
0x143DF	0xD6DB	Text labels
0x147BC	0xDAB8	unknown
0x150B3		Null data
0x15188		unknown
0x15E51		Null data
0x15F34		unknown
0x168D2		Null data
0x168F2		unknown
0x16904		Null data
0x169FF		end of data (rest of file is junk)

Maps

The maps for Repton are not compressed, so each byte will represent a character. However, Repton 2 maps are stored as per Method Two on page 7.

Transporters, Puzzle Pieces and Spirits

The co-ordinates for the Transporters, Puzzle Pieces and Spirits are worked out thus, to give a number between 0 and 31. The screens will be between 1 and 18:

```
x=(X-12)div 4
y=(Y-24)div 4
```

Where X and Y are the values stored, with x and y being the actual co-ordinates.

Z80 format

The ZX Spectrum emulators can store a snapshot of the memory, in many different formats. One of the most common is the .z80 format, which is described here (abridged so that only the information needed to extract Repton data is listed).

offset	Description
0x0000	Header
0x0006	If 0x0006 and 0x0007 are zero, file is version 2 or 3, otherwise 1
0x000C	If bit 5 is set on version 1, data is compressed
0x001E	V1 - Data (as laid out in memory); V2/3 - Second header
0x0022	V2/3 - machine type (0x0: 48K, 0x4:128K, 0x7: +3, 0xC: +2)
0x0056	V2/3, not machine 0x7 – Compressed data, in data blocks
0x0057	V2/3, machine 0x7 – Compressed data, in data blocks

Version 2/3 Data block headers

offset	Length	Description
0x0000	2 bytes	Length
0x0002	1 byte	Page
0x0003	x bytes	Data

Version 2/3 pages

Page	Memory offset	
	128K/+2/+3	48K
3	0xC000 to 0xFFFF	
4		0x8000 to 0xBFFF
5	0x8000 to 0xBFFF	0xC000 to 0xFFFF
8	0x4000 to 0x7FFF	

Compression

The compression method is very simple: it replaces repetitions of at least five equal bytes by a four-byte code ED ED xx yy, which stands for "byte yy repeated xx times". Only sequences of length at least 5 are coded. The exception is sequences consisting of ED's; if they are encountered, even two ED's are encoded into ED ED 02 ED. Finally, every byte directly following a single ED is not taken into a block, for example ED 00 00 00 00 00 00 is not encoded into ED ED ED 06 00 but into ED 00 ED ED 05 00. The block is terminated by an end marker, 00 ED ED 00.

Full details of both the z80 and DSK format can be found on the World Of Spectrum web site (www.worldofspectrum.org).

EGO: Repton 4

Passwords

The passwords for EGO can be found in the main program file, !Repton.ObjCode, from offset 0x168AC to 0x169A3. There are a total of 31 passwords, including the cheat (the last one). There is no terminating character as each password is 8 characters long (31*8=248). Each byte needs to be XOR-ed with 0xFF to get the ASCII code.

Level data

All the EGO Repton 4 levels are stored in the !Repton4.Resources.Allmaps file.

Each map is made up of 19x19 tiles. These are represented in the file unencoded, and as it is laid out as on the screen. However, each tile is represented as a 4-byte word, and the screens are not in screen order.

```
Character position = ((level-1)*361*4)+((y-1)*19*4)+((x-1)*4)
where 1<=level<=30;1<=x<=19;1<=y<=19
```

To get to the appropriate data for the screen, there are 30 bytes in the ObjCode which are offsets, multiplied by the data size, into the Allmaps file. This table can be found at offset 0x165D0, and each screen is 0x5A4 in size. Therefore, the formula you need is:

```
Offset into Allmaps = (byte-1)*0x5A4
```

Each tile is laid out as a 4-byte word. Bytes 0, 1 and 2 are used to store extra information. Where this information is not used, it is generally 0x00 but treat it as undefined. Byte 3 refers to the characters:

0x00: Repton
0x01: Puzzle placement
0x02: Up conveyor
0x03: Down conveyor
0x04: Left conveyor
0x05: Right conveyor
0x06: Android

> *Byte 1 is unknown. Byte 2 is the direction and speed with 0x01, 0x02 & 0x03 representing Left/Right and 0x04, 0x05 & 0x06 representing Up/Down. The higher the number the faster it moves.*

0x07: Gem

> *Byte 1 is how many times to pick up, or how many gems are on the space, depending on how you look at it.*

0x08: Tower/Gem

> *Byte 1 is the same as a Gem*

0x09: Tower/Potion
0x0A: Tree
0x0B: Tower
0x0C: Disappearing tree (i.e. walk into it and it disappears)
0x0D: Blank (or grass, again, depending on how you look at it)
0x0E: Mushroom
0x0F: Black hole

> *Byte 1 can be 0x01 for always there, or other values indicating the length of time until it appears.*

0x10: Transporter

> *Byte 1 is the transporter number, and byte 2 is either 0x00 for a destination or 0x01 for a source.*

0x11: Puzzle piece

> *Byte 1 is the piece number. These are numbered, on a completed puzzle, 1 to 5 across, then 6 to 10 on the second row, etc. to 25.*

0x12: Transporter/Black Hole

> *Bytes 1 and 2 as for the Transporter*

0x13, 0x14 and 0x15: Unused
0x16: Disappearing tree with bonus

> *Byte 1 - extra bonus score x6000. You'll also get an extra life for this tile.*

PC Repton 3 Graphics

The graphics files, *.r3g, used by PC Repton 3 contain all 216 characters in Windows Bitmap format. Each graphic does not have the bitmap header, as they are all the same, and are stored ‘back-to-back’. Details of the order of the 216 characters are given in the PC Repton 3 Editor under Help. Normally, you wouldn’t need to worry about the format, as the editor takes the bitmaps and creates the file. However, if you wanted to edit existing graphics, you would need to split this file back into the bitmaps, which the editor does not do.

Windows Bitmap Layout

This is an abridged version of the format, with only the information required for these files listed. Addresses/offsets use little endian. The numbers in brackets are the actual values used for each bitmap.

Offset	**Description**
0x0000	File Header
0x000E	DIB Header
0x0036	Palette
0x????	Raw bitmap data (see headers for offset)

Offset	**Size**	**Description**
File Header		
0x0000	2	'BM' Identifies it as a BMP (0x4D42)
0x0002	4	Size of the file in bytes (0x00001438)
0x0006	4	Reserved (0x00000000)
0x000A	4	Offset to pixel data (0x00000436)
DIB Header		
0x000E	4	Size of DIB header (0x00000028)
0x0012	4	Bitmap width (0x00000040)
0x0016	4	Bitmap height (0x00000040)
0x001A	2	Colour planes (0x0001)
0x001C	2	Colour depth/bits per pixel (0x0008)
0x001E	4	Compression method (0x00000000)
0x0022	4	Size of the raw bitmap data, i.e. [0x02]-[0x0A] (0x00001002)
0x0026	4	Horizontal resolution (0x00000B12)
0x002A	4	Vertical resolution (0x00000B12)
0x002E	4	Number of colours in the palette (0x00000000)
0x0032	4	Number of important colours - generally ignored (0x00000000)
Rest of data		
0x0036	0x400	Colour palette - BB,GG,RR,0x00 for each of the 256 colours
[0xA]	[0x22]	Bitmap raw data. Rows are padded to multiples of 4 bytes. Each byte represents a colour number into the palette, per pixel. Row 0 is the bottom of the image.

Therefore, in order to reconstitute a bitmap, you will need to supply the first 0x0436 bytes of data, for each bitmap. You can easily see that the format expected is 64x64px at 8bpp.

Repton The Lost Realms

File Layout

BBC	Electron	Usage
0x000000	0x000000	Screen A Data
0x00029D	0x00029D	Screen B Data
0x00053A	0x00053A	Screen C Data
0x0007D7	0x0007D7	Screen D Data
0x000A74	0x000A74	Screen E Data
0x000D11	0x000D11	Screen F Data
0x000FAE	0x000FAE	Character Data
0x002CAE	0x001FFE	EOF

Encoding

Four characters are packed into 3 bytes by only using bits 0 to 5 and spreading them across the 3 bytes:

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

5	4	3	2	1	0	5	4	3	2	1	0	5	4	3	2	1	0	5	4	3	2	1	0
Character 1						***Character 2***						***Character 3***						***Character 4***					

Screen Data

0x000 to 0x275	Map Data (630 bytes) 30x28 – packed as above
0x276 to 0x27B	Password (8 characters packed into 6 bytes, as above)
0x27C to 0x293	Transporter Data for 8 transporters, 6 bytes each packed as above. Stored as source X, Y, destination X, Y. The source X of all transporters is stored in the first 6 bytes (unpacked to 8 co-ordinates), then the source Y, etc.
0x294 to 0x295	Colour Data, 4 bits per colour (11110000 33332222)
0x296 to 0x297	Edit Code (LSB/MSB)
0x298 to 0x299	Bomb Time (LSB/MSB)
0x29A to 0x29B	Freeze time (LSB/MSB)
0x29C bit 7	Map Visibility
0x29C bits 4 to 6	Fungi Rate
0x29C bits 0 to 3	Absorbency Rate

Character Data

The BBC Micro uses 7424 bytes for 58 characters, which equates to 128 bytes each at 16x32 pixels in size. The Acorn Electron uses 4176 bytes for 58 characters, which equates to 72 bytes each at 12x24 pixels in size. Both platforms are stored as MODE 5 screen data.

Characters

00 Boulder 01 Diamond 02 Earth 1

03 Earth 2	04 Time Capsule	05 Skull
06 Blank	07 Wall	08 Wall L
09 Wall R	0A Wall T	0B Wall B
0C Wall TL	0D Wall TR	0E Wall BL
0F Wall BR	10 Wall 2	11 Wall 2 TL
12 Wall 2 TR	13 Wall 2 BL	14 Wall 2 BR
15 Barrier	16 Safe	17 Cage 1
18 Cage 2	19 Door	1A Freeze Pill
1B Balloon	1C Absorbency Pill	1D Egg
1E Key	1F Fungus	20 Time bomb
21 Transporter	22 Crown	23 Repton
24 Spirit 1	25 Spirit 2	26 Spirit 1 frame 2
27 Spirit 2 frame 2	28 Repton Die 1	29 Repton Die 2
2A Monster	2B Monster frame 2	2C Monster frame 3
2D Repton looking left	2E Repton looking right	2F Repton Walking R 1
30 Repton Walking R 2	31 Repton Walking R 3	32 Repton Walking R 4
33 Repton Walking L 1	34 Repton Walking L 2	35 Repton Walking L 3
36 Repton Walking L 4	37 Repton Up 1	38 Repton up 2
39 Egg cracking		

Clones and Similar Games: Ripton

Ripton, which was a clone of Repton, was written by Kenton Price and submitted to A&B Computing magazine. However, they did not dare publish it. These days it can be found on A&B compilation disc images, freely available on the web.

Characters

Full size (16x32px) characters (128 bytes each)

Like BBC Repton 3, the full size characters are stored complete, in MODE 5 format (see page 8).

File	Offset	Length	Descripton
RIPTON	0x3600	0x100	Ripton die characters (2)
RIPTON	0x3E00	0x700	Monster/Ripton/Broken Eggs (14)
RIPTONB	0x0000	0x900	Others (18)

Map size (8x4) characters (8 bytes each)

Again, stored complete in MODE 5 format.

RIPTON1	0x0F00	0x2F0	94 Text characters
	0x11F0	0x010	2 blanks
	0x1200	0x080	16 map characters

Maps

These are not encrypted in any way in Ripton, and only having 16 characters, it means that each half byte can be used for each character position.

File	Offset	Length	Descripton
RIPTON	0x1E00	0x1400	32x32 characters per level, 4 bits per char, 512 bytes per map

Palette

There is no stored palette in Ripton. Instead, it is calculated depending on the level number.

File	Offset	Length	Descripton
RIPTON	0x1BF8	0x1C	Code to change palette. `Col=(level AND 3)+3; If Col=3 then Col=1`

Passwords

File	Offset	Length	Descripton
RIPTON	0x3D00	0xB0	Passwords. 16 bytes each, in screen order – 11 passwords (inc one for screen editor). 1st byte XOR 2nd byte = 1st chr Result XOR next byte = next chr 0x0D terminates password

Time limits

As Repton, the time limits are always 6000 (0x1770)

Clones and Similar Games: HW Repton 3

There are many clones of the various Repton games, written by others for other platforms hereto Superior has not written for. Between 1998 and 2002, Harry Wood wrote and released such a clone of Repton 3 for the PC. It has not been updated since, as Superior wrote and released the official PC versions of the games. You can still find Harry's version at www.harrywood.co.uk/repton3.

File Status

There are two states of the data files – one that is being edited, and one that has been locked. The first four bytes of the file will indicate which state the file is in:

Offset	Length	Description
0x0000	0x04	EDIT or LOCK

The file format will then differ.

EDIT Format

Each screen is held sequentially, and are 0x3DC bytes in length. The data starts immediately after the file status string. For all bytes read, subtract 0x21 to get any meaningful data.

Offset	Length	Description
0x0004	7	Password
0x000B	1	Map visible: 1=yes
0x000C	3	Time Limit – stored as three characters: 100s,10s,1s
0x000F	0x3C1	Map - 31x31, bottom row first
0x03D0	16	Transporter details, 4 bytes each (sx,sy,dx,dy) x 4

Total file size will be 7908 (0x1EE4) bytes.

LOCK format

A locked file is somewhat more complex, as it has been encoded with a lock password, and counter.

Offset	Length	Description
0x0004	15	Lock Password
0x0013	1	Counter start (byte - 0x21)
0x0014		Start of map data

Lock Password

The lock password is held in 15 bytes, but only 8 of them are used. Bytes 1,3,5,7,9,11,13 contain the password characters where the character ASCII code = (byte - (chr_pos - 1)) + 1. The other bytes (0,2,4,6,8,10,12,14) are random, and can be safely disposed of.

LOCKed Maps

Each screen map is interlaced with each other, and the passwords, transporters, time limits and visible flag are interlaced with the map data. Also, the column and row order is interlaced. In addition, it is encoded with the lock password and a counter.

For each byte, the unencoded byte = byte - 0x21 - p_encode - counter.

- p_encode is each successive password character ASCII (to the length of the password, then reset to the beginning), where 60 is taken away until it is 60 or under;
- counter is 0 to 9 inclusive, counting the number of bytes processed (reset to 0 when 10 is reached), and starts with value at 0x0013 minus 0x21.

In addition, if the undecoded byte is '5' (ASCII 0x35), before subtracting 0x21, then following byte is skipped.

The data is stored in this order:
row1,col31,scr1
row1,col31,scr2
to
row1,col31,scr8
row1,col30,scr1
row1,col30,scr2
and then to
row1,col1,scr8

Following the first row of each screen, the password, interlaced with transporter details and time limit are stored:
For the first 4 characters of the password:
pword_chr
trans_sx
trans_sy
trans_dx
trans_dy
(remember, as the map is 'bottom-up', that the y co-ordinates will reflect this)

Then, for characters 5,6 and 7 of the password:
pword_chr
time_limit_chr

And then finally visible flag. The data then continues, in a similar fashion to above:
row2,col31,scr1
to
row31,col1,scr8
without password/transporter/time limit/visible details.

Final Notes

Passwords are padded with ASCII 32 to either 7 (screen password) or 8 (lock password). However, these spaces are skipped on the encoding/decoding, so it will depend on the length of the password up to the first SPACE. Transporter y co-ordinates are as map is laid out (i.e. y=31-byte, as it is 'bottom-up').

Graphics

The game graphics are held, as Bitmaps, within the Windows executable file itself. Therefore, you can either create your own, or write a utility to extract them.

Clones and Similar Games: Bonecruncher

This was produced using information published by David Boddie, and then investigated further by Gerald Holdsworth. So far, this document only covers the BBC Micro and Acorn Electron versions, which are fairly similar.

BONE2/BONE_2

The following refers to the files BONE2 on the BBC Micro disc version (also Master Compact and Play It Again Sam disc version), and BONE_2 (Acorn Electron version).

Offset	**Length**	**Usage**
0x0000	0x1400	Sprites - 40 off, 0x80 bytes each, 16x32px, MODE5 format
0x1400		Code
0x14C9		Unknown
0x15F7	0x0016	Map tile to sprite lookup table - BBC Micro
0x164E	0x0016	Map tile to sprite lookup table - Acorn Electron
0x1???		Code, including:
0x1727		Number of lives (stored at &0A) - BBC Micro
0x173F		Soaps required to complete level (stored at &05) - BBC Micro
0x1774		Number of lives (stored at &0A) - Acorn Electron
0x178C		Soaps required to complete level (stored at &05) - Electron
0x17C3		Invulnerability (code reduces value at &0E by 1) - BBC Micro
0x1809		Invulnerability (code reduces value at &0E by 1) - Electron
0x1853		Infinite Lives (code reduces value at &0A by 1) - BBC Micro
0x189A		Infinite Lives (code reduces value at &0A by 1) - Acorn Electron
0x1CD0		Fozzy Infinite energy (code reduces value at &08 by 1) - BBC
0x1CF7		Fozzy Infinite energy (code reduces value at &08 by 1)-Electron
0x1E31	0x0010	Animation sequences - 4 bytes each for Glook, Monster, Spider and Fozzy - BBC Micro
0x1E58	0x0010	Animation sequences - 4 bytes each for Glook, Monster, Spider and Fozzy - Acorn Electron
0x1FAD	0x0004	Bono walking right animation sequence - BBC Micro
0x1FB1	0x0004	Bono walking left animation sequence - BBC Micro
0x1FB5	0x0002	Bono walking up/down animation sequence - BBC Micro
0x1FD3	0x0004	Bono walking right animation sequence - Acorn Electron
0x1FD7	0x0004	Bono walking left animation sequence - Acorn Electron
0x1FDB	0x0002	Bono walking up/down animation sequence - Acorn Electron
0x206F		Code
0x20D2		LOAD":0.$.SCREEN1"3300 [0x0D] - BBC Micro
0x20E9		DISK [0x0D] - BBC Micro
0x20EE		Data
0x2110		Code
0x22B8		Unknown
0x22??		Code
0x22FB		Unknown
0x2408	0x00F8	Passwords, in reverse level order, each terminated with 0xFF. Add 0x55 to reveal ASCII, or 0x40=SPC
0x2500	0x0C00	Maps storage area for 6 levels (SCREEN1, 2, 3 and 4 are loaded

		here). 0x200 bytes per level
0x3100	0x1F40	Playing screen
0x5040		Code
0x50D8	0x0370	Monster with soap screen – BBC Micro
0x5290		LOAD :0.$.Screen1 3300 [0x0D] - Acorn Electron
0x52A7		DISC [0x0D] - Acorn Electron
0x5448		Unknown
0x587C		Code
0x5900	0x0040	Data - gets moved to &60
0x5940	0x0020	Data - gets moved to &180
0x5960	0x0100	Data - gets moved to &300
0x5980	0x0100	Data - gets moved to &880
0x5A80		Unknown
0x5AE0		Code
0x5B8D		Data

SCREENx

Where x is 1=levels 1 to 6, 2=levels 7 to 12, 3=levels 13 to 18, 4=levels 19 to 24.

Each row is at least 20 bytes in length, with each four bits representing a tile in the map, the lower four bits appearing on the left and the upper four bits on the right. Each special tile requires an additional half byte, so rows with special tiles are longer than 20 bytes in length.

Map size is 40x25 tiles, but each map is padded out with zeros to 0x200 bytes, with the final byte being the actual colour number (3 bits - BGR) of the map.

Tile definitions:

0x0:	Space	0x1:	Horizontal wall
0x2:	Vertical wall	0x3:	Corner wall
0x4:	Cauldron	0x5:	Door/gate
0x6:	Key	0x7:	Earth
0x8:	Trapdoor	0x9:	Sea
0xA:	Glook	0xB:	Skeleton
0xC:	Monster	0xD:	Spider
0xE:	Space (unknown/unused)		

0xF: Special tile - these are always followed by another value which determines which special tile is used:

0x0:	Fozzy	0x1:	Rightward stairs
0x2:	Leftward stairs	0x3:	Upward stairs
0x4:	Downward stairs	0x5:	Bono
0x6:	Volcano	0x7:	Unused

Sprite order

This is the order the sprites appear at the beginning of BONE2 (BBC) or BONE_2 (Electron):

0x01	Horizontal Wall	0x02	Vertical Wall	0x03	Corner Wall
0x04	Sea	0x05	Cauldron	0x06	Rightward Stairs
0x07	Leftward Stairs	0x08	Upward Stairs	0x09	Downward Stairs

0x0A	Gate	0x0B	Key	0x0C	Trapdoor
0x0D	Earth	0x0E	Skeleton 1	0x0F	Skeleton 2
0x10	Glook 1	0x11	Glook 2	0x12	Glook 3
0x13	Monster standing	0x14	Monster 1	0x15	Monster 2
0x16	Spider 1	0x17	Spider 2	0x18	Fozzy standing
0x19	Fozzy 1	0x1A	Fozzy 2	0x1B	Bono dead
0x1C	Bono walking right 1	0x1D	Bono walking right 2	0x1E	Bono walking r 3
0x1F	Bono walking left 1	0x20	Bono walking left 2	0x21	Bono walking l 3
0x22	Bono Up/Down 1	0x23	Bono Up/Down 2	0x24	Bono Yawn
0x25	Bono Sleepy	0x26	Bono Winking	0x27	Bono hand up
0x28	Bono Standing				

Map tile to sprite Lookup Table

This is a lookup table to find the correct sprite from the map tile byte, i.e. a table that translates one table above to the other. This can be found at 0x15F7 in BONE2 (BBC) or 0x164E in BONE_2 (Acorn Electron).

For the special tiles (type 0xF, followed by a second half byte), just add the two together (e.g. Fozzy is 0xF+0x0=0x0F, Bono is 0xF+0x5=0x14, etc.). Bits 5-7 (0x80 to 0xB0) refer to an animated sprite. The animation sequences can be found at offset 0x1E31 in BONE2 (BBC Micro) or 0x1E58 in BONE_2 (Acorn Electron) and are 4 bytes each (each byte is a frame, and references the sprites).

NOTE: The lookup table translates Bono's character (0x14) to a Skeleton (0x0F), and a volcano (0x15) to Earth (0x0D).

Useful Programs

In this section I present some BBC BASIC programs that you may find useful.

Archimedes and Desktop Repton 3 Decoder

This will iterate through the specified directory structure looking for Archimedes Repton 3 or High Res Desktop Repton 3 files. Once found, it will open them and output the passwords and edit codes into a text file.

```
REM>Editcodes
REM
REM Archimedes Repton 3 and Desktop Repton 3 file decoder
REM written by Gerald Holdsworth
REM (c)2013 GJH Software
REM
REM V1.10 16th December 2013
REM
REM This will open all AR3 and DR3 files found in the specified
REM directory and sub directories, open them and display the passwords
REM and edit codes for all the levels.
REM
ONERRORREPORT:PRINT" at line ";ERL:CLOSE#0:END
CLOSE#0
DIM data 130000,data% 512,typebuf% 10
typebuf%?8=13
VDU26,12
PRINT "Archimedes Repton 3 and Desktop Repton 3 password and edit code decoder"
PRINT "written by Gerald J Holdsworth (c)2013 GJH Software"
PRINT "V1.10 17th December 2013"
PRINT
SYS"OS_GBPB",6,,data%
curdir$=FNgetname(data%+2)
REPEAT
 PRINT"Base directory: ";:X=POS:Y=VPOS
 PRINTTAB(X,Y)curdir$
 INPUTTAB(X,Y)""dir$
 IF dir$="" dir$=curdir$
 SYS"XOS_File",5,dir$ TO type%
 IF type%=0 PRINT"Error: "dir$" cannot be found"
 IF type%=1 PRINT"Error: "dir$" is a file"
 IF type%=3 PRINT"Error: "dir$" is an image"
 IF type%>3 PRINT"Error: "dir$" is not a valid path"
UNTIL type%=2
REPEAT
 PRINT"Output directory: ";:X=POS:Y=VPOS
 PRINTTAB(X,Y)dir$
 INPUTTAB(X,Y)""output$
 IF output$="" output$=dir$
 SYS"XOS_File",5,output$ TO type%
 IF type%=0 PRINT"Error: "output$" cannot be found"
 IF type%=1 PRINT"Error: "output$" is a file"
 IF type%=3 PRINT"Error: "output$" is an image"
 IF type%>3 PRINT"Error: "output$" is not a valid path"
UNTIL type%=2
output%=OPENOUT(output$+".R3editcode")
PROCexamine(dir$)
CLOSE#output%
OSCLI"SETTYPE "+output$+".R3editcode Text"
END
:
DEF PROCwriteln(file%,line$)
LOCAL i
line$=line$+CHR$10
FOR i=1 TO LEN(line$)
 BPUT#file%,ASC(MID$(line$,i,1))
NEXT
ENDPROC
:
DEF PROCexamine(dir$)
LOCAL next%,number%,F$,screen,code,password$,x,ptr,m%,chr,P%,Q%
next%=0
WHILE next%<>-1
```

```
 SYS "OS_GBPB",10,dir$,data%,1,next%,63,"*" TO,,,number%,next%
 IF number%>0 THEN
  F$=FNgetname(data%+&14)
  IF data%!16=2 THEN
   PROCexamine(dir$+"."+F$)
  ELSE
   IF (!data% >>> 20)=&FFF AND (data%!8=28832 OR data%!8=102560) THEN
    X=POS:Y=VPOS
    PRINTTAB(X,Y)dir$"."F$"                                              ";TAB(X,Y);
    IF data%!8=28832 PROCwriteln(output%,dir$+"."+F$+" (Archimedes Repton 3/Desktop
Repton 3 Low Res)")
    IF data%!8=102560 PROCwriteln(output%,dir$+"."+F$+" (Desktop Repton 3 High Res)")
    SYS"OS_File",12,dir$+"."+F$,data
    FOR screen=0 TO 7
     code=0
     password$=""
     FOR x=0 TO 419
      ptr=&380+(screen*420)+x
      IF (data?ptr AND 4) THEN code=code+(data?ptr*2) ELSE code=code+data?ptr
     NEXT
     P%=&380+(screen*420)
     IF data%!8=28832 THEN Q%=47:P%+=47 ELSE Q%=46:P%+=44
     ptr=screen*8
     m%=0
     REPEAT
      chr=(data?(ptr+m%) EOR data?P%) AND &1F OR 64
      P%+=Q%
      IF chr>13 THEN chr=chr AND &DF
      IF chr>64 AND chr<128 THEN password$+=CHR$chr
      m%+=1
     UNTIL m%=7 OR chr<65
     PROCwriteln(output%,"Screen "+CHR$(65+screen)+": "+STR$code+" "+password$)
    NEXT
   ENDIF
  ENDIF
 ENDIF
ENDWHILE
ENDPROC
:
DEF FNgetname(addr%)
LOCAL b$
WHILE ?addr%>31
 b$+=CHR$(?addr%)
 addr%+=1
ENDWHILE
=b$
```

HW Repton 3 Decoder

This is a program, for RISC OS, to read in and decode the data files from HW Repton 3, presented here to assist with dealing with these files.

```
REM>Decode
REM
REM Harry Wood Repton 3 file decoder
REM written by Gerald Holdsworth
REM (c)2016 GJH Software
REM
REM V1.00 18th March 2016
REM
REM This will open the specified file, in the specified directory
REM and display all the details about the file, including the raw
REM map data for all the levels.
REM
ONERRORREPORT:PRINT" at line ";ERL:END
REM Change this to point this towards your directory
dir$="HostFS:$.Data.ReptonFile.Repton3."
REM Change this for the file within that directory
file$="Main/rls"
SYS"OS_File",13,file$,,,dir$ TO ,,,,length%
DIM data% length%
SYS"OS_File",12,file$,data%,0,dir$
VDU26,12
PRINT"       File: "file$
s$=CHR$(data%?0)+CHR$(data%?1)+CHR$(data%?2)+CHR$(data%?3)
PRINT"  File status: "s$
```

```
p$=""
ctr%=0
pe%=0
b%=32
DIM
pword$(8),time%(8),t_sx%(8,4),t_sy%(8,4),t_dx%(8,4),t_dy%(8,4),map%(8,32,32),vis%(8)
IFs$="EDIT"THEN
  ptr%=&4
  FORm%=1TO8
    pword$(m%)=""
    FORc%=1TO7
      ue%=FNgetbyte
      IFue%<>32pword$(m%)+=FNc(ue%)
    NEXT
    vis%(m%)=FNgetbyte=1
    time%(m%)=(100*FNgetbyte)+(10*FNgetbyte)+FNgetbyte
    FORy%=1TO31
      FORx%=1TO31
        map%(m%,x%,32-y%)=FNgetbyte
      NEXT
    NEXT
    FORc%=1TO4
      PROCtrans(m%,c%)
    NEXT
  NEXT
ENDIF
IFs$="LOCK"THEN
  FORi%=0 TO 7
    b%=?(data%+5+(i%*2))
    b%=(b%-i%)+1
    IFb%<>32p$+=FNc(b%)
  NEXT
  PRINT"Lock Password: "p$
  ptr%=&13
  ctr%=(data%?ptr%)-&21
  ptr%+=1
  FORx%=1TO31
    FORy%=1TO31
      FORm%=1TO8
        map%(m%,x%,32-y%)=FNgetbyte
      NEXT
    NEXT
    m%=x%
    IFm%<9THEN
      pword$(m%)=""
      time%(m%)=0
      FORc%=1TO7
        ue%=FNgetbyte
        IFue%<>32pword$(m%)+=FNc(ue%)
        IFc%<5PROCtrans(m%,c%)
        IFc%=5time%(m%)+=100*FNgetbyte
        IFc%=6time%(m%)+=10*FNgetbyte
        IFc%=7time%(m%)+=FNgetbyte
      NEXT
      vis%(m%)=FNgetbyte=1
    ENDIF
  NEXT
ENDIF
FORm%=1TO8
  PRINTSTRING$(93,"-")
  PRINT"         Screen: ";m%
  PRINT"Screen password: "pword$(m%)
  PRINT"    Screen time: ";time%(m%)
  PRINT"   Transporters: ";
  FORc%=1 TO 4
    IFFNvaltrans(m%,c%)PRINT;t_sx%(m%,c%)+1;",";t_sy%(m%,c%)+1;" to
";t_dx%(m%,c%)+1;",";t_dy%(m%,c%)+1'STRING$(17," ");
  NEXT
  PRINTSTRING$(17,CHR$127);" Is map visible: ";
  IFvis%(m%)PRINT"Yes"ELSEPRINT"No"
  PRINT"            Map:"
  FOR r%=1 TO 31
    FOR c%=1 TO 31
      PRINTFNzero(map%(m%,c%,r%))" ";
    NEXT
    PRINT
  NEXT
```

```
  IFm%<8THEN
    PRINT"Press SPACE for next map...";
    REPEATUNTILINKEY-99:REPEATUNTILNOTINKEY-99
    PRINTSTRING$(27,CHR$127);
  ENDIF
NEXT
:
END
:
DEFFNc(c%)
IFc%<32ORc%>126c%=32
=CHR$c%
:
DEFFNgetbyte
LOCALue%,p_encode%
ue%=?(data%+ptr%)
ptr%+=1
p_encode=0
IFp$<>""THEN
  IFue%=&35ptr%+=1
  pe%+=1:IFpe%>LENp$pe%=1
  p_encode%=ASCMID$(p$,pe%,1)
  WHILEp_encode%>60:p_encode%-=60:ENDWHILE
  ctr%=(ctr%+1)MOD10
ENDIF
=ue%-&21-p_encode%-ctr%
:
DEFFNzero(x%)=RIGHT$("0"+STR$~(x%),2)
:
DEFFNvaltrans(m%,c%)
LOCALv%
v%=TRUE
IFt_sx%(m%,c%)<0ORt_sx%(m%,c%)>30v%=FALSE
IFt_sy%(m%,c%)<0ORt_sy%(m%,c%)>30v%=FALSE
IFt_dx%(m%,c%)<0ORt_dx%(m%,c%)>30v%=FALSE
IFt_dy%(m%,c%)<0ORt_dy%(m%,c%)>30v%=FALSE
IFv%IFmap%(m%,t_sx%(m%,c%)+1,t_sy%(m%,c%)+1)<>&0Av%=FALSE
=v%
:
DEFPROCtrans(m%,c%)
t_sx%(m%,c%)=FNgetbyte
t_sy%(m%,c%)=30-FNgetbyte
t_dx%(m%,c%)=FNgetbyte
t_dy%(m%,c%)=30-FNgetbyte
ENDPROC
```

Acknowledgements

The information in this document was gathered from various sources, and the following people are directly credited for their work:

BBC/Electron Repton and Repton 2 : Jasper Renow-Clarke
BBC/Electron Repton 3: Neil Crutchlow and Jonathan Marten
Sinclair Repton and Repton 2 : Gerald Holdsworth and Gil Jaysmith
Commodore Repton 3 : Gerald Holdsworth
Desktop Repton (all) : Jasper Renow-Clarke, Gerald Holdsworth and Darren Salt
Archimedes Repton and Repton 2 : Jasper Renow-Clarke and Gerald Holdsworth
BBC/Electron Repton Infinity : David Lodge
EGO: Repton 4 : Gerald Holdsworth and Kris Adcock
Harry Wood's Repton 3 : Gerald Holdsworth
BBC/Electron Bonecruncher : David Boddie and Gerald Holdsworth
BBC Ripton : Gerald Holdsworth
PC Repton 3 (graphics) : Gerald Holdsworth

Also, many thanks go to Richard Hanson (Superior Interactive) and Tim Ellert (ESZ Consulting) for all of their, continuing, help and also to David Boddie for his tip of loading files into the screen memory on a BBC to find the location of the graphics.

Decoding Repton compiled by and ©2017 Gerald J Holdsworth

Help with the Repton formats from Neil Crutchlow, Jonathan Marten, Jasper Renow-Clarke, Darren Salt, David Boddie, Gil Jaysmith, Tim Ellert, Kris Adcock and David Lodge

www.reptonresourcepage.co.uk