RISC OS Sprite Formats compiled by Gerald J Holdsworth

This document describes the format of the various types of RISC OS sprites, collated from the information in the RISC OS Programmer's Reference Manuals (most of it lifted directly from the pages), and from various contributors on the Stardot.com forum. This document has also been updated as I write a Windows application to load in RISC OS Sprite Files and output Windows Bitmap files, and I have included information on how to convert from Sprites to Bitmaps. Also note that this document, like the Windows application, is still under development, so is subject to change.

There have been a few changes to the format of Sprites, taking into account new technologies as they become available. "Old format" refers to Arthur through to RISC OS <3.5, and "new format" refers to RISC OS 3.5 to RISC OS 5.21. However, from RISC OS 5.21, the format was expanded even further (shown as RISC OS 5), and is currently still being developed.

When the sprite data is held in memory, the first word contains the total size of the sprite area, but when saved to a file, this first word is omitted, but the offsets in the next couple of words remain the same. This should be bourn in mind when examining the data in a sprite file.
Hexadecimal notation is given in 'C++' form: i.e. 0xFF means 255 in decimal.

Control Block

Offset	Size	Contents
0x00	4	Size of Sprite Area (not held in file)
0x04	4	Number of sprites in area
0x08	4	Byte offset to first sprite
0x0C	4	Byte offset to first free word (i.e. byte after last sprite)
0x10	xx	Extension words (usually null)

Windows Bitmaps only hold a single image, therefore the above block is only required in order to read in the sprites, and check that the file is a Sprite File.

Sprite Format: Control Block

The offsets given in this control block are relative to the start of each block.

Offset	Size	Contents
0x00	4	Offset to next sprite - effectively, size of sprite data
0x04	12	Sprite name with trailing zeros
0x10	4	Width in words-1
0x14	4	Height in scan lines-1
0x18	4	First bit used (left end of row)

In the old format, the pixels did not necessarily start at the beginning of the row. However, in new format, this should be null, as this 'left hand wastage' is no longer used and the pixel data does start at the beginning of each row.

0x1C 4 Last bit used (right end of row)

Each row should be an exact number of words, so this indicates where the end of the row is. However, I have safely ignored this value when converting to bitmaps, as they also need to have their rows in multiples of 4 bytes.

0x20 4 Offset to sprite image

Where the actual pixel data is. Not always before the transparency data, but always after the palette. By examining this, and the next field, it can be determined if the sprite has a palette.

0x24 4 Offset to transparency mask, or sprite image if no mask

This is where converting to a Windows Bitmap becomes difficult – they are generally not transparent. Some applications take the bottom left pixel of a bitmap image as the transparent colour. With sprites using a separate mask, this is effectively adding an extra colour (e.g. a 17th colour in a 4bpp sprite).

0x28	4	Mode sprite was defined in (old format) or Sprite mode word (new format)
0x2C	xx	Palette data (optional)

Sprite mode word (RISC OS 3.5 and above): offset 0x28

'Bits' refer to the bit number, with bit 31 being the left hand bit, and bit 0 being the right hand bit. This makes it confusing when it comes to look at the left and right hand wastage bits – see later on.

RISC OS 3.5

Bit(s)	Usage
31	Wide mask flag
27-30	Sprite Type
14-26	Vertical DPI - should be 22, 45, 90 or 180

When converting to bitmaps, I have generally ignored these values, as they have no use in Windows Bitmaps. These are all set to the same DPI.

1-13	Horizontal DPI - as above
0	=1

RISC OS 5

Bits	Usage
31	Wide mask flag
27-30	if all set, is a RISC OS 5 sprite
20-26	Sprite Type
8-15	Bits 8-15 of the mode flags
6-7	Y eigen value (0=180DPI, 1=90DPI, etc.)
4-5	X eigen value

Bits 27-30, 16-19, and 0-3 must contain the pattern 0x78000001

Determining mode word format

Is the mode word an unsigned value less than 256?
    Yes: It's a mode number
Is the bottom bit zero?
    Yes: It's a pointer to a Mode Selector Block.
Are bits 27-30 all set to 1?
    Yes: It's a RISC OS 5 style sprite mode word
    No: It's a RISC OS 3.5 style sprite mode word

Sprite Types

0: Old mode (see below)

1bpp masks, palette not supported by RISC OS 3.5:

1: 1bpp image (2 colour)

2: 2bpp image (4 colour)

2bpp bitmaps are not supported by most applications in Windows, so when converting from sprite to bitmap, I decided to up-scale 2bpp sprites to 4bpp bitmaps, which caused a few headaches of their own!

3: 4bpp image (16 colour)

4: 8bpp image (256 colour)

8bpp sprites come in three varieties – those without palettes, those with partial palettes (i.e. 64 colours) and those with full palettes (256 colours). See later on for a further explanation.

5: 16bpp image (32K colour)

These have no palettes, as the Red, Green and Blue components are held in 5 bits each of the 16 bits. There is a 16th which can be safely ignored (it is generally zero anyway).

6: 32bpp image (16M colour)
Again, there is no palette. Instead, like 16bpp, the Red, Green and Blue components are given, per pixel. This time, each component is 8 bits long, each, with a 'spare' 8 bits, which can be safely ignored (there are some exceptions – see later).

7:	CMYK
8:	24bpp image
9:	JPEG
10:	16bpp 5:6:5 TBGR
11-14:	Reserved
15:	Invalid (used to id RISC OS 5 sprite mode words)
RISC OS 5:	
16:	16bpp 4:4:4:4
17:	4:2:0 YCbCr
18:	4:2:2 YCbCr
19-127:	Reserved

Mode Flags

With RISC OS 5 sprites, bits 8 – 15 of the mode data gives bits 8 – 15 of the mode flags, which are:

Bit:	Meaning
8:	Full resolution interlaced mode
9:	Greyscale palette
10-11:	Reserved

Bits 12-13	Family	Bit 14	Bit 15	Meaning
0	RGB	0	0	TBGR
		1	0	TRGB
		0	1	ABGR
		1	1	ARGB
1	Misc	0	0	KYMC
		1	0	Reserved
		0	1	
		1	1	
2	YCbCr	0	0	ITU-R BT.601 full
		1	0	ITU-R BT.601 vid
		0	1	ITU-R BT.709 full
		1	1	ITU-R BT.709 vid

Screen Modes (number of colours)

This is used when the sprite type is 0, hence the mode is <256. However, only the following modes have been defined, along with the number of colours that mode is capable of:

0:2	1:4	2:16	3:4	4:2	5:4	6:2	7:16	8:4	9:16
10:256	11:4	12:16	13:256	14:16	15:256	16:16	17:16	18:2	19:4
20:16	21:256	22:16	23:2	24:256	25:2	26:4	27:16	28:256	29:2
30:4	31:16	32:256	33:2	34:4	35:16	36:256	37:2	38:4	39:16
40:256	41:2	42:4	43:16	44:2	45:4	46:16	47:256	48:16	49:256
50:2	51:4	52:16	53:256	54+:n/a					

From this data, a translation table can be built up to give the appropriate bpp for the given mode (actually, my translation table gave the mode flag rather than the bpp).

Palette Data

If a sprite has a palette, the palette data immediately follows the sprite header. Each palette entry is two words long, allowing for flashing colours to be stored. The colours are stored in the standard 0xBBGGRR00 format. Windows Bitmap palettes are four bytes long, and stored 0x00RRGGBB.

As has been mentioned, 256 colour sprites do not necessarily have 256 palette entries. The reason being that 256 colour modes are treated differently from the others. For this reason, when converting to Windows Bitmap, I used a standard palette when the 8bpp sprite either had no palette or less than 256 colours, and the given palette when there was a full one given. However, the following information is given should you wish to tackle this headache.

Instead of using the standard 16 entry physical colour table, there are two systems which are used by different commands. The internal format is the less easy to use of the two. In it, the bits are structured as follows:

Bit	Meaning
0-3	Bits 0-3 of palette index
4	Red bit 3 (high)
5	Green bit 2
6	Green bit 3 (high)
7	Blue bit 3 (high)

Where the palette index (0 - 15) controls which VIDC palette entry is used, but with some bits of the palette entry then being overridden by the top 4 bits of the memory byte. With the default palette setting, this becomes:

Bit	Meaning
0	Tint bit 0 (red + green + blue bit 0)
1	Tint bit 1 (red + green + blue bit 1)
2	Red bit 2
3	Blue bit 2
4	Red bit 3 (high)
5	Green bit 2
6	Green bit 3 (high)
7	Blue bit 3 (high)

Each primary colour has 4 bits of intensity, but the two least significant bits (the tint bits) are shared between the three colours. Therefore, some intensities of a primary colour (for example, red) can only be obtained at the expense of adding in a certain amount of grey.

The second form for 256 colours, which is used by some commands is structured as follows:

Bit	Meaning
0	Red bit 2
1	Red bit 3 (high)
2	Green bit 2
3	Green bit 3 (high)
4	Blue bit 2
5	Blue bit 3 (high)
6	Tint bit 0 (red + green + blue bit 0)
7	Tint bit 1 (red + green + blue bit 1)

The number of entries in the palette can be worked out thus:
palette_entries = (min(offset_to_image,offset_to_mask) - 44) div 8

The full definition for 32bpp is:

Bits	Meaning
0-6	Value showing how colour was programmed
7	Supremacy bit
8-15	Red

16-23	Green
24-31	Blue

with bits 0-7:

Value	Meaning
0-15	Actual colour (BBC): bit 0: Red, bit 1: Green, bit 2: Blue, bit 3: flash
16	Defined by giving RGB
17-18	Flashing colour by RGB

Palette Files

Should you choose to use external palette files, saved from !Paint in RISC OS, their format is fairly straightforward. They are stored as a sequence of 'VDU19' codes, so each palette entry is six bytes long and is 0x13 <colnum> 0x10 RR GG BB

Pixel format

The width of the sprite, in pixels, can be worked out thus:
pixel_width = (word_width_minus_one*32 + right_bit + 1 - left_bit) / bpp

Converting from sprite pixel format to bitmap pixel format is relatively straightforward. For 1bpp and 8bpp, it is a straight copy. For 4bpp, the upper and lower nibbles need to be swapped. For 2bpp, as has been mentioned, these need to be up scaled to 4bpp, and bits need to be moved around. The following algorithm sorts this:
With every byte ‘t’ read from the Sprite:

```
        t2=((t AND 0xC0) >> 6) + (t AND 0x30)
        t=((t AND 0x0C) >> 2) + ((t AND 0x03) << 4)
```

And then stored into the bitmap data in the order ‘t’ followed by ‘t2’. You will also need to make allowances for the fact that the pixel data is up to double the size of the original.

16 and 32 bits per pixel format

Old format: Each pixel is a group of bytes per character bits. It is not necessarily 4 pixels per word. There will be left hand wastage and right hand wastage (defined by the offsets at 0x18 and 0x1C).

Colour	16bpp sprites	32bpp sprites
Red	bits 0-4	bits 0-7
Green	bits 5-9	bits 8-15
Blue	bits 10-14	bits 16-23
Reserved	bit 15 (should be 0)	bits 24-31 (should be 0)

When converting to Windows Bitmaps, the Red and Blue components will need to be swapped. However, the palettes can be left unchanged (of course, without the extraneous bytes, as a bitmap palette is four bytes per entry).

Transparency Mask

Old format: Mask is same size as sprite image, and same bits refer to each pixel. Each bits of each pixel must either be all set (solid) or all cleared (transparent).
New format: Mask is 1 bit per pixel; each row of mask bits begins word aligned with the remainder of the 32 bits being padded as set. A set bit means solid, while a cleared bit means transparent.

Best way that I have found of checking whether a transparency mask is new or old format is just to compare the data size of both the pixel area and the transparent mask – if they are the same size, then it is old format, while a smaller transparent mask size means new format.

Bitmaps are generally are not transparent. Some applications use the Alpha byte in the palette (or, for 32bpp, in the data itself) to indicate an opacity setting, with 0xFF being fully transparent, while other applications use the bottom left pixel (i.e. first in the data) as the transparent colour. This may mean that you might need to up-scale a sprite's bpp to add in an extra colour to use as the background. One way around this, and is my chosen method, is to determine how many colours are actually used in the pixel data – then you can just add another, if there are enough entries spare, for the transparent colour. Otherwise, you will need to up-scale it, and the headaches that that might present.

Windows Bitmap Format

In order to convert from one to the other, you will need to know both formats. Therefore, for completeness, here is the format for the majority of Windows Bitmap images:

Offset	Size	Usage
0x00	2	'BM' – identifier for Windows Bitmaps
0x02	4	Size of file in bytes
0x06	2	Reserved (actual use depends on application creating it)
0x08	2	Reserved (actual use depends on application creating it)
0x0A	4	Offset to pixel data
0x0E	4	Size of DIB header (should be 40 bytes)
0x12	4	Width of bitmap in pixels
0x16	4	Height of bitmap in pixels
0x1A	2	Number of colour planes (should be 1)
0x1C	2	Bits per pixel (1, 4, 8, 16, 24 or 32)
0x1E	4	Compression method (0=none)
0x22	4	Size of raw bitmap data (i.e. [0x02]-[0x0A])
0x26	4	Horizontal resolution (pixel per metre – should be 0x0B12)
0x2A	4	Vertical resolution (as above)
0x2E	4	Number of colours in the palette (0 means 2^[0x1C])
0x32	4	Number of important colours (generally ignored, so should be 0)
0x36	xx	Palette Data (if any)

Then follows the pixel data, each row padded to four bytes. Also note that where the Sprite pixel data is stored top down (i.e. the top row appears first in the data), Windows Bitmaps are bottom down (i.e. the top row appears last in the data).

Gerald Holdsworth
13th July 2017
www.geraldholdsworth.co.uk