IFF Icon Data Decoded

Background information:

The image data for the IMAG chunks are encoded using a modified version of RLE aka Run Length Encoding. It is a bitstream rather than a byte stream composed of RLE bytes and image data bytes.

Perhaps, the IFF icon image data storage format is a trade-off between complexity and practicality. Because applying the RLE encoding to chunky data is much more efficient than applying the same encoding to planar data the storage size is much smaller and thus much more practical for Amiga computers which in the early days had very limited storage space. However, the storage format is complex and difficult to implement so bit shifting and masking is needed for encoding & decoding.

Because of the complexity of it let's look at some sample image data in the Hex Editor as well as Some Pseudo Roman Numeral Notation to help explain how to interpret and decode the image data. The large I's and small I's are used to indicate the transition from input byte to the next byte in the sequence. The colored I's indicate which bits in the bitstream are shared between the output bytes.

The actual bytes for bitstream in a Hex Editor:

Calculator.info

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000480  FF FF FE 01 00 3F FF FF FC 00 00 1F FF FF F8 00  ÿÿþ..?ÿÿü...ÿÿø.
00000490  00 0F FF FF F0 00 46 4F 52 4D 00 00 06 A2 49 43  ..ÿÿð.FORM...¢IC
000004A0  4F 4E 46 41 43 45 00 00 00 06 29 2D 00 11 00 2A  ONFACE....)-...*
000004B0  49 4D 41 47 00 00 02 C3 00 10 03 01 01 05 02 8D  IMAG...Ã........
000004C0  00 2A 81 06 80 3F C3 FE 2F 01 FF 8B FC 3E 20 FF  .*..€?Ãþ/.ÿ‹ü> ÿ
000004D0  0F 50 80 82 42 F9 80 0C 22 10 7B 8A 08 11 08 86  .P€‚Bù€.".{Š...†
000004E0  F3 07 F8 80 42 5F E3 FF 23 F8 C0 24 37 C9 01 44  ó.ø€B_ãÿ#øÀ$7É.D
000004F0  11 19 04 F4 07 F8 80 82 53 FE A7 FD 42 4B 4A 56  ...ô.ø€‚Sþ§ýBKJV
00000500  C4 A5 4A 5A 46 20 88 C6 21 3D 40 08 31 04 A8 FB  Ä¥JZF ˆÆ!=@.1.¨û
00000510  48 92 96 95 2D 2A 5A 54 B1 88 22 32 0C 4F 50 02  H'–•-*ZT±ˆ"2.OP.
00000520  0C 41 2A 3F 52 20 C5 A9 89 53 18 A6 25 6C 18 82  .A*?R Å©‰S.¦%l.‚
00000530  2F E3 00 13 D4 03 A3 10 4A 64 A5 8A 5B 12 96 2D  /ã..Ô.£.Jd¥Š[.–-
00000540  2C 5A 58 A6 0C 41 11 90 62 7A 80 20 62 09 4C 94  ,ZX¦.A..bz€ b.L"
```

81 06 80 3F C3 FE 2F 01 FF 8B FC 3E 20 FF 0F 50 80 82 42 F9 80 0C 22 10 7B 8A 08 11 08 86 F3 07 F8 80 42 5F E3

The first 9 input bytes from the bitstream are: 0x81, 0x06, 0x80, 0x3F, 0xC3, 0xFE, 0x2F, 0x01, 0xFF. Note that even the RLE control bytes are encoded in the bitstream. For the 2-byte pairs that indicate byte runs of equal bytes there is a repeating pattern. In the bitstream the first RLE byte is just a byte. It represents 8 bits so it is shown in black. Since there is no previous byte no bit shifting is needed to retrieve the byte value. So, in a sense, the first byte is not part of the encoded bitstream. But starting with the next byte in the input sequence is where the bitstream encoding actually begins.

For 2-byte pairs there will be 8 bits followed by image data which is encoded by depth. For example, if depth is 5 then each 8 bit value is followed by a 5 bit value sandwiched in between the RLE bits. The pattern of encoded data repeats for each of the 2-byte pairs – 8 bits, 5 bits, 8 bits, 5 bits, 8 bits, 5 bits, and so on. Bit shifting and masking should be used to retrieve the output byte values including the RLE control bytes and image data bytes. For RLE copy bytes which are strings of single bytes that are not equal to each other the pattern of storage bits is different. First there is an 8 bit RLE value followed by however many image bits needed by depth. So, with depth of 5 to store 4 image bytes will be 8 bits, 5 bits, 5 bits, 5 bits, 5 bits. The 4 image bytes of 5 bits each only occupy 20 bits so they easily fit within just 3 bytes of input data in the bitstream with 4 unused bits remaining at the end. Those 4 remainder bits will be used with the next byte in the sequence to retrieve the next 8 bit RLE value. It isn't wasted.

Pseudo Roman Numeral Notation:

RLE Formula: If (value > 128) then value = (256-value+1), If (value < 128) then value = value+1

129 06 128 63 195 254 47 01 255
10000001, 00000110, 10000000, 00111111, 11000011, 11111110, 00101111, 00000001, 11111111,

IIIIIIII - IIIIIIII - IIIIIIII - IIIIIIII - IIIIIIII - IIIIIIII - IIIIIIII - IIIIIIII - IIIIIIII

10000001, 00000, 11010000, 00000, 11111111, 00001, 11111111, 00010, 11110000, 00011,
[129]128 0 [208]49 0 [255]2 1 [255]2 2 [240]17 3

First 9 bytes decoded: {128,0} {49,0} {2,1} {2,2} {17,3}

Decoded: 198 bytes will be written to the output buffer.

Mask Values for RLE & image data:

For masking the mask for the 8 bit RLE bytes is 0xFF (255) in binary: 11111111. For image data with depth of 5, for example Mask = (1<<depth)-1. Since total number of colors by depth is (1<<depth) which for depth of 5 is 32, the corresponding mask will be 32-1. 31 in binary: 00011111. The 1's in each mask value are the number of bits that will be retrieved from the bitstream after bit shifting into place.

mask = 0xFF (255) or mask = (1<<depth) – 1.

The bitbuffer container & the bit count:

For bit shifting and masking a 32 bit long container called the "bitbuffer" is used. Only up to 16 bits of input data will be used to retrieve RLE bytes and image bytes at any given time. For reasons of keeping track of the position in the current input byte and for deciding how many bits to shift to the right to get the bits in the correct position for masking the "bit count" and "bit index" values are used. The bit count represents the current number of bits which is always less than 8. The bit index is (bit count + 8). If at any time bit count is > depth then a "bit adjust" (bits -= depth) is needed to keep the bit count in range.

In order to decode the image data load each input byte and apply bit shifting and masking as needed to retrieve the RLE bytes and image bytes till the end of the input byte buffer is reached at which time the number of total bytes in the output byte buffer should be equal to width x height of the icon image itself.

Basic decoding methods:

Here are some of the basic code sample methods to load and decode the bytes from the bitstream.

To load an RLE byte from the input bitstream do this:

```
if(bits < 8) then bitbuf = (bitbuf<<8)|*(srce++); //load byte
```

then apply the “bit index” bits += 8; //bit index

To get the 8 bit RLE byte value do this:

```
val = (bitbuf>>(bits-8))&0xFF;  //get byte with mask
```

then apply the “bit adjust” bits -= 8; //bit adjust

Note that “val” is “unsigned char val”. It’s a byte value. After we retrieve the value from the bitstream then we need to evaluate the value.

To evaluate the byte value do this:

if (val <= 127) then (copy = val + 1); //RLE control byte. This is the number of single bytes to be copied.

else if (val > 128) then loop = (256 - val +1); //RLE control byte. This is the number of equal bytes to write to the output byte buffer.

Inside the processing while loop if “copy” (single bytes) or “loop” (equal bytes) are > 0 then we perform the operations to write the appropriate data to the output byte buffer.

For “loop” values to load a byte do this:

```
if (loop) {

if(bits < depth) bitbuf = (bitbuf<<8)|*(srce++); //load byte

bits += 8; //bit index
```

To get the actual byte value do this:

```
val = (bitbuf>>(bits-depth))&mask; //get byte with mask
```

Then setup a for-loop to process the equal bytes to be written to the output byte buffer…

```
for (i=0; i<loop; i++) {

dest[out+i] = val; }
```

After that we need to reset the various location variables such as…

```
idx++; //RLE loop byte, bits -= depth; //bit adjust,  out += loop;  loop = 0;

} //end if loop
```

For "copy" values to load & process a byte do this:

Note: Because we are not just copying the same value repeatedly as with "loop" values for "copy" values representing a string of not-equal bytes we must do everything within a dedicated for-loop.

```
if (copy) {

   for (i=0; i<copy; i++) {

   if(bits < depth)

   { bitbuf = (bitbuf<<8)|*(srce++); //load byte, bits += 8; //bit index

   } //end if bits

   val = (bitbuf>>(bits-depth))&mask; //get byte with mask

   dest[out+i] = val;

   idx++; //RLE copy bytes,

   bits -= depth; //bit adjust

}  //end for loop

out += copy;

copy = 0;

} //if copy
```

Conclusion:

Although the storage format for the encoded image data in the IMAG chunks is complex it serves its purpose in keeping the size of the icon file small. If we look at the size of RLE encoded planar data in the icon image stored as an ILBM file the image data seems larger. So the encoded bitstream method is more efficient at storing the chunky bytes of data for the images. It would have been easier to just copy the RLE encoded planar data from the body of the ILBM directly to the IMAG chunk but then we would lose the efficiency of the compression method. In some ways the complexity of the storage format is really daunting, but at the same time the practicality of it is also evident by allowing more efficient storage.

The method to encode chunky bytes of image data to store it in the bitstream of data is just the reverse. However, bit shifting is needed to encode the data in the bitstream. For decoding the IFF icon image data the programmer should start by writing a "DecodeRLE" algorithm for decoding run length encoding. For encoding the IFF icon image data likewise start by writing an "EncodeRLE" algorithm. For both then add necessary bit shifting and masking. That's all there is to encoding and decoding the image data for IMAG.