是否将当前网页翻译成中文 网页翻译 关闭
翻译

A Primer on the ETC2 format

Motivation

I was working on a personal project that could benefit from texture compression on the GPU, and wanted to support both Desktop and Mobile. When I looked into what the compression formats were like, I was surprised to find very little good documentation for how the mobile formats actually worked. After spending a significant amount of time taking notes on the information I found and a couple hours pouring over the original ETCPACK implementation of the compressor by Ericsson himself, (available on GitHub) I decided that doing a writeup to share with other people who might want all the information in one place would be a good thing.

ETC the same old spiel

If you somehow found your way to this document without an understanding of what ETC is or what it might be useful for, I will give you a quick rundown.

ETC (Ericsson Texture Compression) is a texture compression format originally designed on the principal that the Human ocular system (your eyes) is much more perceptive to differences in luminance (brightness) than chrominance (color). Because of this it makes sense to break down an image into smaller regions (blocks) and store a base color for each region along with smaller offsets in luminance for each pixel in the region. This is a lossy form of compression, but it does a passable job in enough cases.

This is super useful in hardware accelerated graphics for two reasons.

1. ETC compression achieves a 4:1 compression on RGBA data, (I am interested in the application to RGBA so this whole writeup is based on it) meaning you can fit four times as much texture data into the same amount of VRAM. With the usage of higher resolution textures in games today, most high budget games can have gigabytes, or even terabytes of texture data. This means that getting more data onto the GPU can lead to loading less frequently or moving things around less, both big wins.
2. One of the major limiting factors in GPU performance is memory bandwidth. Compressed texture data actually works on the GPU in a way such that you use less memory bandwidth to fetch texels from a texture that is stored in a compressed format. This means better performance, another big win.

ETC1

The original specification of ETC compression is based on an older compression format called PACKMAN, and was originally called iPACKMAN (improved PACKMAN). This was later renamed to ETC, and when the specification was updated to ETC2, the original ETC became ETC1.

ETC1 is actually pretty simple in its format. This is really nice because ETC2 decoders are backwards compatible with ETC1 encoded data. So if you don't want to do a lot of bit banging, (we'll get into this later) you don't really have to, you can just implement the simple 444 Mode and Differential Mode of the ETC1 standard and boom, you get 4:1 reduction in file size with some artifacting in specific cases.

ETC1 makes some simple breakdowns of the image data into more managable chunks. (blocks) A block is simply defined as a 4X4 region of pixels which the ETC1 algorithm will compress into a smaller code for storage. This means that an image has to have dimensions that are multiples of 4 for the compression to work. (Pad the image if it is not a multiple of 4) Each block gets reduced to two 64 bit payloads, one storing color data, and one storing alpha data. Moving from 16*4=64 bytes of data per block to 2*8=16 bytes of data per block gets us that 4:1 compression number.

> NOTE: I say here that ETC1 stores a 64 bit payload for alpha, but it is important that ETC1 doesn't actually support any formats that store alpha data.
>
> This writeup is concerned with ETC2 and we are talking about ETC1 in the capacity that an ETC2 decoder is able to handle it. An ETC2 decoder will handle ETC1 encoded data with the alpha data without any complaints in the COMPRESSED_RGBA8_ETC2_EAC or COMPRESSED_SRGB8_ALPHA8_ETC2_EAC formats.

In the ETC1 modes each of these 4X4 pixel blocks is broken down into two sub-blocks that each have their own base color and what is referred to as a codeword. (really just an index) The codeword for each sub-block is used with a 2-bit pixel index that is stored for each pixel to look up an offset in what is called a codebook. (really just a table or 2d array) This offset is used to offset (duh) the base color of the block in the luminance direction for each pixel in that sub-block. Offsetting in the luminance direction is just a fancy way of saying that we are going to add the same value to all Red, Green, and Blue channels.

How this is all stored.

- The base colors are stored together using 8 bits for each channel. (24 bits total)
- Each codeword is stored using 3 bits. (6 bits total)
- Each pixel index is stored as 2 bits. (32 bits total)

是否将当前网页翻译成中文 网页翻译 关闭

The eagle-eyed among you might notice 2 things here.

1. 3-bit codeword + 2-bit pixel index: the codebook must have 32 entries, and is probably 8X4 in dimension. Both of which are correct.
2. 32+24+6 =/= 64. (The size of the payload we are compressing this block down to) Where do the extra two bits go?

The extra two bits are each used as flags.

- One bit is used to indicate if the sub-blocks are oriented horizontally (4X2) or vertically. (2X4)
- The other bit is used to indicate which one of the ETC1 Modes is used to encode the base colors of the block.

How is this laid out in memory?

$byte_0$	$byte_1$	$byte_2$	$byte_3$	$byte_4$	$byte_5$	$byte_6$	$byte_7$
red	green	blue	cw_0 | cw_1 | d | f	pixel indexes			

fig: top row: byte boundries, bottom row: layout for ETC1 encoded block

- red: red color channel data (8 bits)
- green: green color channel data (8 bits)
- blue: blue color channel data (8 bits)
- cw_0: codeword 0 (3 bits)
- cw_1: codeword 1 (3 bits)
- d: differential flag (1 bit)
- f: flip flag (1 bit)
- pixel indexes: 16*2 bits for pixel indexes (32 bits)

You might ask how we are storing two colors in only one color worth of channels, but we will go over that in a bit. It is handled differently depending on the value of the diff bit.

For now, let's get a look at what that codebook looks like:

	0	1	2	3	4	5	6	7
0	2	5	9	13	18	24	33	47
1	8	17	29	42	60	80	106	183
2	-2	-5	-9	-13	-18	-24	-33	-47
3	-8	-17	-29	-42	-60	-80	-106	-183

fig:ETC1 codebook
horizontal index is codeword, vertical is pixel index

Those of you who have done the reading might notice something interesting about this codebook: it isn't laid out like the ones in a lot of the other resources available online. Why? Because for some reason the people that wrote those other resouces decided to put the entries in an order that looks pretty instead of the order that the entries actually appear in the hardware. (if you can't tell, I wish this weren't the case, so I'm fixing it here) If you look in the

comments of Ericsson's original implementation or in some small comments here and there in other resources online it specifically states that the table should be laid out this way in memory so that the first bit of the pixel index can be used to indicate sign. </END OF SMALL RANT>

Now that we've got the basics out of the way, on to how these payloads are decoded in the ETC1 modes we mentioned earlier.

444 Mode

The difference between this mode and the next is how they decode the color channels. The 444 mode does nothing special, it treats each color as RGB4, so each channel has color 0 packed in the high nibble and color 1 packed in the low nibble.

$byte_0$		$byte_1$		$byte_2$	
C_{0r}	C_{1r}	C_{0g}	C_{1g}	C_{0b}	C_{1b}

fig:top row: byte boundries, bottom row: 444 mode color layout
each box here is 4 bits totalling 24 bits

Simple, right? Yes. After the decoder unpacks these bits, it then expands them out to 8 bits. It does this via a method called bit copying. Simply it puts bits as high as they will go into the byte, then copies in the left over low bits from the high end of the bits being copied in.

$byte_0$	
C_0	C_0

fig:444 mode color expansion copy

each box here is 4 bits filling the 8 bits of the

是否将当前网页翻译成中文 网页翻译 关闭

The next step after this is to add the offset. Using the codeword for th pixel we are decoding, we look for the offset value and add it to each of the channels for this pixel to get the final color of that pixel. Blammo, we have decoded a pixel using the ETC1 444 Mode. (code follows)

```
//payload: the data in the encoded block value
//image: the data of the image to be written with [y][x] layout
//x: x value of the top left corner of the block to be decoded
//y: y value of the top left corner of the block to be decoded
void decode444(u8[] payload, u8[][] image, u32 x, u32 y) {
  //first pull out each of the color's color channel data from the payload
  u8 c0r4 = payload[0]|7,4|;
  u8 c0g4 = payload[1]|7,4|;
  u8 c0b4 = payload[2]|7,4|;

  u8 c1r4 = payload[0]|3,0|;
  u8 c1g4 = payload[1]|3,0|;
  u8 c1b4 = payload[2]|3,0|;

  //we are doing the color extraction in two parts here, in the ETC2 modes we won't
  //then use bit copying to extend the colors to RGB8
  u8 c0r = c0r4 << 4 | c0r4;
  u8 c0g = c0g4 << 4 | c0g4;
  u8 c0b = c0b4 << 4 | c0b4;

  u8 c1r = c1r4 << 4 | c1r4;
  u8 c1g = c1g4 << 4 | c1g4;
  u8 c1b = c1b4 << 4 | c1b4;

  //retreive the codewords from the payload
  u8 codeword0 = payload[3]|7,5|;
  u8 codeword1 = payload[3]|4,2|;

  //retreive the pixel indexes into a more convenient form
  u8[] pixelIndexes = u8[16];
  for(u8 a = 0; a < 4; a++)
  for(u8 b = 0; b < 4; b++) {
    pixelIndexes[a*4 + b] = payload[4 + a]|7 - b*2, 6 - b*2|;
  }

  //check if the sub-blocks are horizontal or vertical
  if(!payload[3]|0,0|) { //flip bit indicates horizontal sub-blocks
    //iterate over the pixels in each sub-block and set their final values in the image data
    for(u8 a = 0; a < 2; a++)
    for(u8 b = 0; b < 4; b++) {
      i8 codebookValue = codebookETC1[codeword0][pixelIndexes[a*4 + b]];
      u32 imageX = 4*(x + b);
      u32 imageY = y + a;
      image[imageY][imageX] = clamp(0, c0r + codebookValue, 255);
      image[imageY][imageX + 1] = clamp(0, c0g + codebookValue, 255);
      image[imageY][imageX + 2] = clamp(0, c0b + codebookValue, 255);

      codebookValue = codebookETC1[pixelIndexes[(a + 2)*4 + b]][codeword1];
      imageY = imageY + 2;
      image[imageY][imageX] = clamp(0, c1r + codebookValue, 255);
      image[imageY][imageX + 1] = clamp(0, c1g + codebookValue, 255);
      image[imageY][imageX + 2] = clamp(0, c1b + codebookValue, 255);
    }
  } else { //flip bit indicates vertical sub-blocks
    for(u8 a = 0; a < 4; a++)
    for(u8 b = 0; b < 2; b++) {
      i8 codebookValue = codebookETC1[codeword1][pixelIndexes[a*4 + b]];
      u32 imageX = 4*(x + b);
      u32 imageY = y + a;
      image[imageY][imageX] = clamp(0, c0r + codebookValue, 255);
      image[imageY][imageX + 1] = clamp(0, c0g + codebookValue, 255);
      image[imageY][imageX + 2] = clamp(0, c0b + codebookValue, 255);

      codebookValue = codebookETC1[pixelIndexes[a*4 + b + 2]][codeword0];
      u32 imageX = imageX + 8;
      image[imageY][imageX] = clamp(0, c1r + codebookValue, 255);
      image[imageY][imageX + 1] = clamp(0, c1g + codebookValue, 255);
      image[imageY][imageX + 2] = clamp(0, c1b + codebookValue, 255);
    }
  }
}
```

For reference, the pixel indexes are stored in this order for all ETC modes:

	X+	→			
Y+	0	1	2	3	
↓	4	5	6	7	
	8	9	10	11	
	12	13	14	15	

*fig.*pixel index layout diagram

not anything too special here

是否将当前网页翻译成中文 网页翻译 关闭

Differential Mode

Differential mode works in basically the same way as 444 mode, but the way that it unpacks the color data from the incomming payload is different. Instead of being stored as RGB4 data, the first color is stored as RGB5 data and the second color is stored as 3-bit offsets to the first color. This gives a higher precision if the base colors of the two sub-blocks are similar.

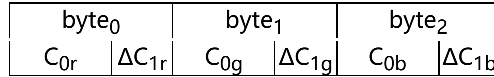


fig:differential mode color layout

each left box of a pair here is 5 bits and its partner is 3, totalling 24 bits

The docoder unpacks these values into 6 bytes, then shifts the differential bytes up then down by 5 bits to extend the sign of the differential value into the upper bits and put the value into two's compliment. This allows the offset to be anywhere in the range [-4, 3]. Once this is done color 1 is expanded from the RGB5 values to RGB8 values using bit copying, and color2 is expanded from the RGB5 values with the deltas added on to RGB8 using bit copying as well.

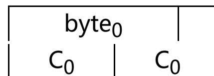


fig:differential mode color expansion copy

the top 5 bits of the byte are filled with the 5 bits from the RGB5 data, then the lower bits are filled with the upper bits of the data again. this is a better example of bit copying than 444 since it doesn't line up perfectly

Just like in 444 Mode, the next step is to add the offset to the base color value. Again, the codeword for the block and the pixel index for the pixel being decoded are used to look up the offset value from the codebook and then that value is added to each channel of the base color. (code follows)

```
//cs5: the RGB5 color data already extracted from the payload to determine mode by overflow
void decodeDifferential(u8[] payload, u8[][] image, u32 x, u32 y, cs5[][]) {
  //extend color 0 color channels via bit copying
  u8 c0r = cs5[0][0] << 3 | cs5[0][0] >> 2;
  u8 c0g = cs5[0][1] << 3 | cs5[0][1] >> 2;
  u8 c0b = cs5[0][2] << 3 | cs5[0][2] >> 2;

  //extend color 1 color channels via bit copying
  u8 c1r = cs5[1][0] << 3 | cs5[1][0] >> 2;
  u8 c1g = cs5[1][1] << 3 | cs5[1][1] >> 2;
  u8 c1b = cs5[1][2] << 3 | cs5[1][2] >> 2;

  //note from here out this is identical to the 444 decode
  u8 codeword0 = payload[3]|7,5|;
  u8 codeword1 = payload[3]|4,2|;

  u8[] pixelIndexes = u8[16];
  for(u8 a = 0; a < 4; a++)
  for(u8 b = 0; b < 4; b++) {
    pixelIndexes[a*4 + b] = payload[4 + a]|7 - b*2, 6 - b*2|;
  }

  if(!payload[3]|0,0|) {
    for(u8 a = 0; a < 2; a++)
    for(u8 b = 0; b < 4; b++) {
      i8 codebookValue = codebookETC1[codeword0][pixelIndexes[a*4 + b]];
      u32 imageX = 4*(x + b);
      u32 imageY = y + a;
      image[imageY][imageX] = clamp(0, c0r + codebookValue, 255);
      image[imageY][imageX + 1] = clamp(0, c0g + codebookValue, 255);
      image[imageY][imageX + 2] = clamp(0, c0b + codebookValue, 255);

      codebookValue = codebookETC1[pixelIndexes[(a + 2)*4 + b]][codeword1];
      imageY = imageY + 2;
      image[imageY][imageX] = clamp(0, c1r + codebookValue, 255);
      image[imageY][imageX + 1] = clamp(0, c1g + codebookValue, 255);
      image[imageY][imageX + 2] = clamp(0, c1b + codebookValue, 255);
    }
  } else {
    for(u8 a = 0; a < 4; a++)
    for(u8 b = 0; b < 2; b++) {
      i8 codebookValue = codebookETC1[codeword1][pixelIndexes[a*4 + b]];
      u32 imageX = 4*(x + b);
      u32 imageY = y + a;
      image[imageY][imageX] = clamp(0, c0r + codebookValue, 255);
      image[imageY][imageX + 1] = clamp(0, c0g + codebookValue, 255);
      image[imageY][imageX + 2] = clamp(0, c0b + codebookValue, 255);

      codebookValue = codebookETC1[pixelIndexes[a*4 + b + 2]][codeword0];
      u32 imageX = imageX + 8;
      image[imageY][imageX] = clamp(0, c1r + codebookValue, 255);
      image[imageY][imageX + 1] = clamp(0, c1g + codebookValue, 255);
      image[imageY][imageX + 2] = clamp(0, c1b + codebookValue, 255);
```

```
        image[imageY][imageX + 2] = clamp(0, c1b + codebookValue, 255);
      }
    }
  }
```

是否将当前网页翻译成中文 网页翻译 关闭

That's it for the ETC1 decoding modes, hopefully you are still with me here.

Alpha decode

I think it is worth saying this again, but alpha isn't supported under ETC1 normally, but we are really focusing on ETC2 here. With that said, alpha is stored in a method very similar to the normal ETC1 444 Mode, but since there is only one channel, it is simplified a bit. But don't worry, the simplification in decode is made up for in the complexity of the codebook.

So I guess we will take a look at the codebook first:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	-3	-3	-2	-2	-3	-3	-4	-3	-2	-2	-2	-2	-3	-1	-4	-3
1	-6	-7	-5	-4	-6	-7	-7	-5	-6	-5	-4	-5	-4	-2	-6	-5
2	-9	-10	-8	-6	-8	-9	-8	-8	-8	-8	-8	-7	-7	-3	-8	-7
3	-15	-13	-13	-13	-12	-11	-11	-11	-10	-10	-10	-10	-10	-10	-9	-9

fig.alpha codebook base definition
oh... it get's worse

You might be sitting in your seat right now thinking, "Well that isn't too bad, it's only twice the size of the codebook for ETC1." But oh ye nieve soul, this is only the base definition of the alpha codebook, we'll call it A for posterity. Now hold on to your butts.

0	..15	..31	..47	..63	..79	..95	..111	..127	..143	..159	..175	..191	..207	..223	..239	..255
..3	0A	1A	2A	3A	4A	5A	6A	7A	8A	9A	10A	11A	12A	13A	14A	15A
..7	0(-A-1)	1(-A-1)	2(-A-1)	3(-A-1)	4(-A-1)	5(-A-1)	6(-A-1)	7(-A-1)	8(-A-1)	9(-A-1)	10(-A-1)	11(-A-1)	12(-A-1)	13(-A-1)	14(-A-1)	15(-A-1)

fig.alpha codebook full definition
the horizontal index here is codeword, and the vertical index is pixel index

That's a big codebook. So... the codeword is obviously 8 bits and the pixel index is 3 bits. Note that each cell of this codebook is defined as an interger multiple of the base definition A. Now some of you might ask, "Why are the first two cells wasted on 0 values? You only really need one column of 0 values to get that offset mode." I know this was my first reaction when I figured out what this codebook looked like. After thinking about it for a while, I am pretty sure the answer comes down to the difference between hardware and software implementations.

In software, it is significantly faster to store all of your values in one big array so that using two indexes you can look up a value in one memory operation. In hardware, everything kind of happens at the same time, so it makes more sense to make a couple smaller tables that each have multiple indexes to save space on the silicon. This leads to why there are 16 columns of 0 values. I speculate that this isn't one big table in the hardware, but one smaller table that looks like our old friend A. This would mean that the hardware decodes the 8 bit codeword as two 4 bit values, the upper nibble is a multiplier, and the lower nibble is an index into A along with the pixel index, where the high bit of the pixel index is used to indicate sign of the table lookup.

Something like this:

pixel index			codeword	
	s	pi	mul	ti

fig. top row: byte boundries, bottom row: bit layout of alpha codeword and pixel index

- s: sign bit for value looked up from the table A (1 bit)
- pi: pixel index used to look up the value from table A (2 bits)
- mul: multiplier applied to the value looked up from the table A (4 bits)
- ti: table index used to look up the value from table A (4 bits)

NOTE: by storing the negative value in A, the behavior $m*(-A[cw_l, pi|1,0|] - 1)$ actually does the work of translating the negative two's compliment value to a positive unsigned value. So the negative sign there is probably actually indicating bit-level negation.

Honestly this information about how the hardware actually treats the values isn't particularly useful to

somebody looking to understand how compressing something into the ETC
to understand the reasoning behind why certain things are done the way th
the reasoning behind the decision to store so many zero values.

是否将当前网页翻译成中文 网页翻译 关闭

Okay, now that we've got an understanding of how the codebook looks, and probably works, let's take a look at how the alpha payload is laid out.

byte$_0$	byte$_1$	byte$_2$	byte$_3$	byte$_4$	byte$_5$	byte$_6$	byte$_7$
base	cw	pixel indexes					

fig: top row: byte boundries, bottom row: bit layout of alpha code

- base: the base alpha of the block (8 bits)
- cw: codeword used as a lookup into the codebook (8 bits)
- pixel indexes: 16*3bit pixel indexes used as a lookup into the codebook (48 bits)

the block is not broken down into sub-blocks in alpha compression, the whole block has one base value

Ah, that is refreshingly simple. The method for decoding is pretty straightforward too. For each pixel in the order described above, extract the pixel index from the list. Then use it with the codeword to look up an offset from the codebook. Lastly, add the offset to the base value and clamp to [0,255] to get the final alpha value of that pixel.

That wraps up the definition of how alpha is stored in the ETC2 RGBA formats. (code follows)

```
void decodeAlpha(u8[] payload, u8[][] image, u32 x, u32 y) {
  //extract base alpha value for block from payload
  u8 baseAlpha = payload[0];

  //extract codeword value for block from payload
  u8 codeword = payload[1];

  //extract pixel indexes for block from payload into a more convenient format
  u8[] pi = u8[16];
  pi[0] = payload[2]|7,5|;
  pi[0] = payload[2]|4,2|;
  pi[0] = payload[2]|1,0| << 1 | payload[3]|7,7|;
  pi[0] = payload[3]|6,4|;

  pi[0] = payload[3]|3,1|;
  pi[0] = payload[3]|0,0| << 2 | payload[4]|7,6|;
  pi[0] = payload[4]|5,3|;
  pi[0] = payload[4]|2,0|;

  pi[0] = payload[5]|7,5|;
  pi[0] = payload[5]|4,2|;
  pi[0] = payload[5]|1,0| << 1 | payload[6]|7,7|;
  pi[0] = payload[6]|6,4|;

  pi[0] = payload[6]|3,1|;
  pi[0] = payload[6]|0,0| << 2 | payload[7]|7,6|;
  pi[0] = payload[7]|5,3|;
  pi[0] = payload[7]|2,0|;

  //traverse the pixel array to set the final alpha values
  for(var a = 0; a < 4; a++)
  for(var a = 0; a < 4; a++) {
    image[y + a][4(x + b) + 3] = clamp(0, baseAlpha + codebookAlpha[codeword][pi[a*4 + b]], 255);
  }
}
```

NOTE: in an ETC2 encoded block, the alpha payload comes before the color payload.

Next we will move on to the more complicated ETC2 extension modes.

ETC2

ETC1 has some noticable artifacts in 2 main cases.

1. When the chrominance values of a sub block are not distributed near the base color or along the luminance direction of the base color.
2. When the chrominance values of a block gradually change across a range of values. (small gradients)

In order to combat these two situations it was proposed that the original ETC1 format be extended with new modes. But an important point of contention was retaining the 4:1 compression ratio. This meant that no extra bits could be added to the compressed payload to indicate the new modes. A maybe not so simple method to do this was found.

In ETC1 differential mode, some of the possible combinations of base color and offset result in overflow. In an ETC1 decoder, these overflowed values are simply clamped and nothing interesting happens, but an ETC2 decoder

uses this overflow to indicate which of the ETC2 modes is used to encode th

是否将当前网页翻译成中文 网页翻译 关闭

Overflow in the Red channel indicates that ETC2 T-Mode is used, over ETC2 H-Mode is used, and overflow in the Blue channel indicates that ETC2 Planar Mode is used.

Here is a quick description of how this mechanism works. (code follows)

```
void decodePayload(u8[] payload, u8[][] image, u32 x, u32 y) {
  if(payload[3]|1,1|) {
    u8 c0r5 = payload[0]|7,3|;
    u8 c0g5 = payload[1]|7,3|;
    u8 c0b5 = payload[2]|7,3|;

    i8 c1rd = (payload[0]|2,0| << 5) >> 5;
    i8 c1gd = (payload[1]|2,0| << 5) >> 5;
    i8 c1bd = (payload[2]|2,0| << 5) >> 5;

    i8 c1r5 = c0r5 + c1rd;
    i8 c1g5 = c0g5 + c1gd;
    i8 c1b5 = c0b5 + c1bd;

    if(c1r5 > 31 || c1r5 < 0) {
      decode59T(payload, image, x, y);
    } else if(c1g5 > 31 || c1g5 < 0) {
      decode58H(payload, image, x, y);
    } else if(c1b5 > 31 || c1b5 < 0) {
      decode57P(payload, image, x, y);
    } else {
      decodeDifferential(payload, image, x, y [[c0r5, c0g5, c0b5], [c1r5, c1g5, c1b5]]);
    }
  } else {
    decode444(payload, image, x, y);
  }
}
```

59-bit T-Mode

Welcome to the first of three modes defined in the ETC2 specification. The reason that I mentioned that you might not want to implement these in a compressor before is because they use a significant amount of bit banging to get data into and out of the compressed payloads. So stap in, we're in for a little bit of a ride.

First we will get right into it and take a look at the 59-bit T-mode payload layout compared to the normal diff mode payload layout.

<table>
<tr><td colspan="2">$byte_0$</td><td colspan="2">$byte_1$</td><td colspan="2">$byte_2$</td><td colspan="5">$byte_3$</td><td>$byte_4$</td><td>$byte_5$</td><td>$byte_6$</td><td>$byte_7$</td></tr>
<tr><td colspan="2">red</td><td colspan="2">green</td><td colspan="2">blue</td><td colspan="2">cw_0</td><td>cw_1</td><td>d</td><td>f</td><td colspan="4">pixel indexes</td></tr>
<tr><td></td><td>R_{0a} | | R_{0b}</td><td>G_0</td><td>B_0</td><td>R_1</td><td>G_1</td><td>B_1</td><td colspan="2">C_a</td><td>d</td><td>C_b</td><td colspan="4">pixel indexes</td></tr>
</table>

fig: top row: byte boundries, middle row: standard block bit layout, bottom row: bit layout of 59-bit T-mode

- R_{0a}: high bits of red channel of $color_0$ (2 bits)
- R_{0b}: low bits of red channel of $color_0$ (2 bits)
- G_0: green channel of $color_0$ (4 bits)
- B_0: blue channel of $color_0$ (4 bits)
- R_1: red channel of $color_1$ (4 bits)
- G_1: green channel of $color_1$ (4 bits)
- B_1: blue channel of $color_1$ (4 bits)
- C_a: high bits of codeword (2 bits)
- d: diff bit (must be 1) (1 bit)
- C_b: low bit of codeword (1 bit)
- pixel indexes: 16*2-bit indexes for each pixel in the block (32 bits)

dark grey blocks are unable to be used to store information

This probably clues you in to a couple different things right off the bat.

- There are 4 bits in what would normally be the red channel that are unable to be used for data. This is because the normal differential interpretation of those bits must overflow. The only way we can control that happening is by setting the 4 bits that are used to store the R_0 channel and then setting the rest of the bits to assure overflow.
- The bits for each of the color channels are 4 bits long. The base colors for this mode must be stored as RGB4, and this is actually the case for both T-Mode and H-Mode.

> NOTE: when storing the 4 bits for R_{0a} into the first byte of the payload, it makes sense to store them and then alter the other bits into an overflow state. The simplest way to do this would probably be to have a precomputed table for the 16 possible combinations of the 4 bits to be stored.

This isn't really that complicated when we get right down to it. From here you just extract the color channels and codeword out of the payload each into their own bytes, then use bit copying to extend the 4 bit color channels out to

8 bits just like in 444 Mode.

是否将当前网页翻译成中文 网页翻译 关闭

Now for the special sauce of T-Mode. The pixel indexes are not used a... lookup into a codebook, just the codeword is. We'll see what the pixel indexes are used for in a bit here, but first let's get a look at the codebook for ETC2.

0	1	2	3	4	5	6	7
3	6	11	16	23	32	41	64

fig:ETC2 codebook
the top row here is the codeword used to access the codebook

Well, this is pretty simple compared to the alpha codebook, but the reason for that is becase the pixel index is used as a lookup into a color table instead of the codebook, so there are only 3 bits worth of address to use for looking into this codebook. The data retrieved from this codebook is used as an offset to one of the base colors in the luminance direction to generate additional color values.

"How are the colors in this table determined?" you might ask. Let's take a look.

Table Color	value
T_0	$color_0$
T_1	$color_1$ + codebook[codeword]
T_2	$color_1$
T_3	$color_1$ - codebook[codeword]

fig: color table definition for 59-bit T-Mode
color+number here means adding that number to each channel of the color. (luminance offset) the resulting value is clamped to the [0,255] range

Now we have all the informationg about the implementation of the 59-bit T-Mode, all that is left to do here is to use the pixel indexes as indexes into this table to decode the color values of each pixel in the order that was shown earlier. (code follows)

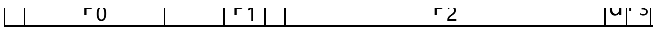
```
void decode59T(u8[] payload, u8[] image, u32 x, u32 y) {
  u8[][] colors = u8[4][3];
  //extract color channels from payload and expand using bit copying
  u8 colors[0][0] = payload[0]|4,3| << 6 | payload[0]|1,0| << 4 | payload[0]|4,3| << 2 | payload[0]|1,0|;
  u8 colors[0][1] = payload[1]|7,4| << 4 | payload[1]|7,4|;
  u8 colors[0][2] = payload[1]|3,0| << 4 | payload[1]|3,0|;

  u8 colors[2][0] = payload[2]|7,4| << 4 | payload[2]|7,4|;
  u8 colors[2][1] = payload[2]|3,0| << 4 | payload[2]|3,0|;
  u8 colors[2][2] = payload[3]|7,4| << 4 | payload[3]|7,4|;

  //extract codeword from payload
  u8 codeword = payload[3]|3,2| << 1 | payload[3]|0,0|;

  //generate offset colors
  u8 colors[1][0] = clamp(0, colors[2][0] + codebookETC2[codeword], 255);
  u8 colors[1][1] = clamp(0, colors[2][1] + codebookETC2[codeword], 255);
  u8 colors[1][2] = clamp(0, colors[2][2] + codebookETC2[codeword], 255);

  u8 colors[3][0] = clamp(0, colors[2][0] - codebookETC2[codeword], 255);
  u8 colors[3][1] = clamp(0, colors[2][1] - codebookETC2[codeword], 255);
  u8 colors[3][2] = clamp(0, colors[2][2] - codebookETC2[codeword], 255);

  //set colors in image data
  for(u8 a = 0; a < 4; a++)
  for(u8 b = 0; b < 4; b++) {
    u8 pi = payload[4 + a]|7 - 2*b, 6 - 2*b|;

    u32 imageY = y + a;
    u32 imageX = 4*(x + b);
    image[imageY][imageX] = colors[pi][0];
    image[imageY][imageX + 1] = colors[pi][1];
    image[imageY][imageX + 2] = colors[pi][2];
  }
}
```

That is all there is to T-Mode, next we will take a look at H-Mode which is very similar.

58-bit H-Mode

T-Mode stored 59 bits into the differential mode payload if you took the time to count or connected the dots. 58-bit H-Mode, as it's name suggests, stores 58 bits into the differential code. Let's take a look at the layout for how it does this:

$byte_0$	$byte_1$	$byte_2$	$byte_3$				$byte_4$	$byte_5$	$byte_6$	$byte_7$
red	green	blue	cw_0	cw_1	d	f	pixel indexes			



是否将当前网页翻译成中文 网页翻译 关闭

fig:top row: byte boundries, middle row: standard block bit layout, bo
H-mode

- P_0: part 0 of the 58-bit H-Mode block (7 bits)
- P_1: part 1 of the 58-bit H-Mode block (2 bits)
- P_2: part 2 of the 58-bit H-Mode block (16 bits)
- P_3: part 3 of the 58-bit H-Mode block (1 bit)

> NOTE: Remember, this mode is signalled by the red channel not having overflow, but the green channel having overflow. As such when the bits from part 1 are packed into byte one during encoding, the first bit of that byte must be chosen so that the rest of the byte will not overflow when unpacked by an ETC2 decoder. Similarly, when inserting the 4 bits from parts 1 and 2 into the second byte, the rest of that byte's bits must be chosen as to avoid overflow. If you built a nice lookup table for doing this in the T-Mode decode, it can be used here too. (Seriously. Just make the table! It only has 16 entries and is by far the fastest way to do this.)

"Wow... that diagram doesn't tell us much about what is in each of those blocks." You're right. Honestly though, making a diagram that showed the internal breakdown would look pretty messy, so we are going to make two diagrams. Get ready for diagram 2. Here. We. Go.

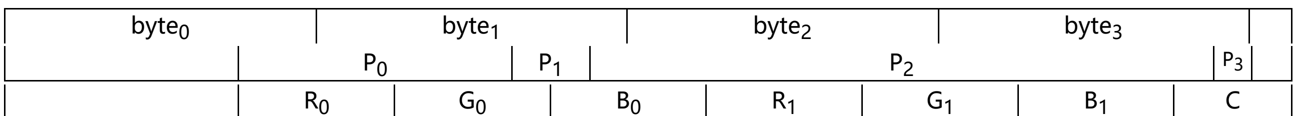


fig: top row: byte boundries, middle row: de-fragmented 58-bit H-Mode data unpacked from differential code, bottom row: bit layout of 58-bit H-Mode data

- R_0: red channel of $color_0$
- G_0: green channel of $color_0$
- B_0: blue channel of $color_0$
- R_1: red channel of $color_1$
- G_1: green channel of $color_1$
- B_1: blue channel of $color_1$
- C: codeword

Now you might be sitting there saying, "Wait a minute, there is a little something hanging off the end there. Where does that come from?" While 58-bit H-Mode does only store 58 bits in the differential mode payload, it works in the same manner that 59-bit T-Mode does. This means that it needs 59 bits of data to decode. So, again, where does the extra bit come from? The answer is something called the "ordering trick."

Data can actually be stored by ordering things in different manners, and that is taken advantage of here to eek out one more bit from the 58 that are stored in the differential mode payload. This is done by comparing the 12 bits of $color_0$ to the 12 bits of $color_1$. If $color_0$ is greater, we get a 1, otherwise 0, and boom there we have our extra bit which is just ord into the low bit of the codeword.

From here it is again pretty simple. We extract the color channel data and codeword each into their own bytes from the payload, then expand the color channels from RGB4 to RGB8 using bit copying just like in 444 Mode. After that we use the codeword to look up the offset value from the ETC2 codebook. (ETC2 uses the same codebook for all modes, so reference the table in the T-Mode section) Finally we use the base colors and offset to construct a color table:

Table Color	value
T_0	$color_0$ + codebook[codeword]
T_1	$color_0$ - codebook[codeword]
T_2	$color_1$ + codebook[codeword]
T_3	$color_1$ - codebook[codeword]

fig: color table definition for 58-bit H-Mode

color+number here means adding that number to each channel of the color. (luminance offset) the resulting value is clamped to the [0,255] range

To finish the decode, we just traverse the pixels in the order indicated before and use the pixel index stored in the payload to look up the final color from this color table. That's all for 58-bit H-Mode. (code follows)

```
void decode58H(u8[] payload, u8[][] image, u32 x, u32 y) {
  //extract color data and codeword from payload
  u16 c0 = payload[0]|6,0| << 5 | payload[1]|4,3| << 3 | payload[1]|1,0| << 1 | payload[2]|7,7|;
  u16 c1 = payload[2]|6,0| << 5 | payload[3]|7,3|;
  u8 codeword = payload[3]|2,2| << 2 | payload[3]|0,0| << 1 | c0 > c1;

  //format color data into RGB8 using bit copying
```

是否将当前网页翻译成中文 网页翻译 关闭

```
    u8 c0r = c0|11,8| << 4 | c0|11,8|;
    u8 c0g = c0|7,4| << 4 | c0|7,4|;
    u8 c0b = c0|3,0| << 4 | c0|3,0|;

    u8 c1r = c1|11,8| << 4 | c1|11,8|;
    u8 c1g = c1|7,4| << 4 | c1|7,4|;
    u8 c1b = c1|3,0| << 4 | c1|3,0|;

    //create color table from base colors
    u8[][] colors = u8[4][3];
    colors[0][0] = c0r + codebookECT2[codeword];
    colors[0][1] = c0g + codebookECT2[codeword];
    colors[0][2] = c0g + codebookECT2[codeword];

    colors[1][0] = c0r - codebookECT2[codeword];
    colors[1][1] = c0g - codebookECT2[codeword];
    colors[1][2] = c0g - codebookECT2[codeword];

    colors[2][0] = c1r + codebookECT2[codeword];
    colors[2][1] = c1g + codebookECT2[codeword];
    colors[2][2] = c1g + codebookECT2[codeword];

    colors[3][0] = c1r - codebookECT2[codeword];
    colors[3][1] = c1g - codebookECT2[codeword];
    colors[3][2] = c1g - codebookECT2[codeword];

    //extract pixel indexes then set colors in image data
    for(u8 a = 0; a < 4; a++)
    for(u8 b = 0; b < 4; b++) {
      u8 pi = payload[4 + a]|7 - 2*b, 6 - 2*b|;

      u32 imageY = y + a;
      u32 imageX = 4*(x + b);
      image[imageY][imageX] = colors[pi][0];
      image[imageY][imageX + 1] = colors[pi][1];
      image[imageY][imageX + 2] = colors[pi][2];
    }
  }
```

Let's move on the the final mode; Planar Mode.

Planar Mode

Planar Mode works quite a bit differently than the other modes do. It doesn't even have a codebook. The reason for this is because planar mode is designed to be able to replicate blocks that have a gradient change from one color to another. The other encoding methods have a hard time reproducing these blocks, and you get block-edge artifacts in the compressed image. So let's get right into it and look at how the data is packed into the differential mode payload:

$byte_0$	$byte_1$	$byte_2$	$byte_3$				$byte_4$	$byte_5$	$byte_6$	$byte_7$
red	green	blue	cw_0	cw_1	d	f	pixel indexes			
P_0	P_1	P_2	P_3		d		P_4			

fig: top row: byte boundries, middle row: standard block bit layout, bottom row: 57-bit Planar Mode layout

- P_0: part 0 of 57-bit Planar Mode block (7 bits)
- P_1: part 1 of 57-bit Planar Mode block (7 bits)
- P_2: part 2 of 57-bit Planar Mode block (2 bits)
- P_3: part 3 of 57-bit Planar Mode block (8 bits)
- d: diff bit (must be 1)
- P_4: part 4 of 57-bit Planar Mode block (33 bits)

> NOTE: Remember, this mode is signalled by the red and green channels in the normal differential mode code not having overflow, and the blue channel having overflow. When inserting the 7 bits into the first two bytes, the first bit of those two bytes must be set so as to avoid overflow. Similarly, when inseting the 4 bits from P_2 and P_3 into the third byte, the other 4 bits of that byte must be set so that the value overflows. I even made the table for you. (See appendix 2)

Ah, we've run into another one of these layouts that doesn't fit into one diagram well. Here we go with diagram numero dos.

$byte_0$	$byte_1$	$byte_2$	$byte_3$	$byte_4$	$byte_5$	$byte_6$	$byte_7$	
	P_0	P_1	P_2	P_3	P_4			
	R_0	G_0	B_0	R_H	G_H	B_H	R_V	G_V	B_V

fig: top row: byte boundries, middle row: de-fragmented 57-bit Planar Mode data unpacked from differential code, bottom row: bit layout of 57-bit Planar Mode data

- R_0: base color red channel (6 bits)

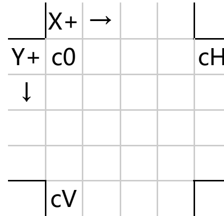
是否将当前网页翻译成中文 网页翻译 关闭

- G_0: base color green channel (7 bits)
- B_0: base color blue channel (6 bits)
- R_H: horizontal color red channel (6 bits)
- G_H: horizontal color green channel (7 bits)
- B_H: horizontal color blue channel (6 bits)
- R_V: vertical color red channel (6 bits)
- G_V: vertical color green channel (7 bits)
- B_V: vertical color blue channel (6 bits)

There you have it, you should probably notice two big things staring you in the face from this last diagram.

1. No pixel indexes here.
2. There are three colors here stored in RGB676

The pixel indexes are not needed here because the final decode in planar mode is just a simple interpolation between these colors to fill the block. According to the Ericsson's ETCPACK implementation, the colors should generally be chosen as such for best results:



So, to finish up we just need to expand these values to RGB8 using bit copying like we have in the past. Then we interpolate to decode the pixels in the block in the order that was showed before. I will leave this interpolation for the code section. (code follows)

```
void decode57P(u8[] payload, u8[][] image, u32 x, u32 y) {
  //extract the different color channels into bytes
  u8 c0r6 = payload[0]|6,1|;
  u8 c0g7 = payload[0]|0,0| << 6 | payload[1]|6,1|;
  u8 c0b6 = payload[1]|0,0| << 5 | payload[2]|4,3| << 3 | payload[2]|1,0| << 1 | payload[3]|7,7|;

  u8 cHr6 = payload[3]|6,2| << 1 | payload[3]|0,0|;
  u8 cHg7 = payload[4]|7,1|;
  u8 cHb6 = payload[4]|0,0| << 5 | payload[5]|7,3|;

  u8 cVr6 = payload[5]|2,0| << 3 | payload[6]|7,5|;
  u8 cVg7 = payload[6]|4,0| << 2 | payload[7]|7,6|;
  u8 cVb6 = payload[7]|5,0|;

  //use bit copying to extend the colors to RGB8
  u8 c0r = c0r6 << 2 | c0r6 >> 4;
  u8 c0g = c0g7 << 1 | c0g7 >> 6;
  u8 c0b = c0b6 << 2 | c0b6 >> 4;

  u8 cHr = cHr6 << 2 | cHr6 >> 4;
  u8 cHg = cHg7 << 1 | cHg7 >> 6;
  u8 cHb = cHb6 << 2 | cHb6 >> 4;

  u8 cVr = cVr6 << 2 | cVr6 >> 4;
  u8 cVg = cVg7 << 1 | cVg7 >> 6;
  u8 cVb = cVb6 << 2 | cVb6 >> 4;

  //set pixel color values in image via interpolation
  for(u8 a = 0; a < 4; a++)
  for(u8 b = 0; b < 4; b++) {
    u32 imageX = 4*(x + b);
    u32 imageY = y + a;

    image[imageY][imageX] = clamp(0, (b*(cHr - c0r) + a*(cVr - c0r) + 4*c0r + 2) >> 2, 255);
    image[imageY][imageX + 1] = clamp(0, (b*(cHg - c0g) + a*(cVg - c0g) + 4*c0g + 2) >> 2, 255);
    image[imageY][imageX + 1] = clamp(0, (b*(cHb - c0b) + a*(cVb - c0b) + 4*c0b + 2) >> 2, 255);
  }
}
```

Well, that's all folks. We have covered all of the different decoding modes of the ETC2 specification. Hopefully you found this helpful.

Afterward

This document, despite being a product of my interest in how the ETC2 format is encoded focuses mostly on the manner in which the different ETC2 encodings are decoded. The reason for this is partially because I haven't actually implemented an encoder yet, but also because if you are looking to build your own encoder it is more important to know how the values will be decoded than how you should encode them.

Thanks for your time, and hopefully you found this document useful.

是否将当前网页翻译成中文 网页翻译 关闭

Appendix 1:

The || operator used in code blocks in this document

Because the syntax of bit manipulation features of many languages differ, the exercise of translating the bit banging used in this specification is left up to the reader. In order to simplify the appearance of code and generalize it more, the || (double pipe) operator is used to signify array type access at the bit level with automatic shift down. Generally:

given a is an byte
a|c,b| where c > b and b >= 0
gives the bit string from 2^c .. 2^b automatically shifted down into the least significant bits
e.g.
given the binary value 0x99 or 10011001 called a
then a|4,2| is the bits for 2^4, 2^3, and 2^2 as a bit string shifted down to the least significant places
so a|4,2| gives 0x06 or 00000110

Appendix 2:

Lookup Table for inserting 4 bits into a byte in a manner that overflows under differential mode decoding.

insert	write	overflow
0x0	0x04	-4
0x1	0x05	-3
0x2	0x06	-2
0x3	0x07	-1
0x4	0x0c	-3
0x5	0x0d	-2
0x6	0x0r	-1
0x7	0xeb	32
0x8	0x14	-2
0x9	0x15	-1
0xa	0xf6	32
0xb	0xf3	33
0xc	0x1c	-1
0xd	0xf9	32
0xe	0xfa	33
0xf	0xfb	34

- insert: nibble to insert into byte while ensuring overflow
- write: byte to write that holds the nibble to insert and overflows
- overflow: value the decode is expected to get during the overflow calculation

Method for ensuring non overflow of byte after insertion of lower 7 bits under differential mode decoding.

To ensure that no over/underflow happens when writing 7 bits into the byte of a color channel, set the leading bit of that byte to the opposite of the leading bit of the 7-bit value.
i.e.
given the 7-bit value to write called a
colorByte = ~a|6,6| << 7 | a|6,1|;

This works because the 3-bit two's compliment number can only produce the values [-4,3].

If the first bit of the 7-bit value is 1, setting the first bit of the byte to zero makes the 5 bit base value the decoder sees somewhere in the range [8,15] and no 3-bit two's compliment value can put any of those values outside the range [0,31].

Similarly, if the first bit of the 7-bit value is 0, setting the first bit of the byte to one makes the 5 bit base value the decoder sees somewhere in the range [16,23] and again the 8-bit two's compliment value cannot put any of these values outside the range [0,31].

Thus this rule ensures these values never over/underflow.

written July 2019 by Nic Johnson : nicjohnson7@yahoo.com

是否将当前网页翻译成中文 网页翻译 关闭