LERC Byte Stream Specification

Blob Header
Mask
Pixel Values

Endianess: Little Endian

Remark: This is the structure in memory. The encoding is done into a memory blob, which can then be saved to disk or transmitted over the network etc.

Multiple bands: Here we describe the encoding of one band. For multiple bands the different single band blobs get concatenated to one. The bit mask is only stored with the first band, and omitted for the other bands.

LERC Blob Header

Item	Format	Size
Format Identifier String	Char[6] = “Lerc2 “	6
Format Version (= 3)	Int	4
Checksum	UInt	4
Image Height in pixel	Int	4
Image Width in pixel	Int	4
Number of valid pixels	Int	4
Micro block size (e.g., 8)	Int	4
Blob size in bytes	Int	4
Lerc datatype	Int	4
MaxZError	Double	8
zMin = min image value	Double	8

zMax = max image value	Double	8

Bit Mask

The valid / invalid bits are represented as a bit stuffed byte array with 8 bits per byte representing 8 pixels. The size of this byte array is then (width * height + 7) / 8. This bit mask is only encoded for the first band and if needed, meaning 0 < number of valid pixels < width * height. If all pixels are valid or all invalid the mask is not encoded. The first element is an integer having the size of the mask blob in bytes. If the mask is not encoded, it is 0.

Item	Format	Size
Mask blob size in bytes	Int	4

The bit mask byte array gets run length encoded as follows: If 5 or more consecutive bytes have the same value (255 or 0 being the most common), a negative short integer equal to the number of same values plus this byte value is encoded (2 + 1 bytes). If less than 5 consecutive bytes have the same value, then a positive short integer equal to the number of different (or not same stretch) bytes is encoded, together with the array of different bytes (2 + n bytes). We use the short integer -32768 to mark the end of the encoded mask blob.

Note that the bit mask, if it gets encoded at all, takes usually much less space than the data itself. Typically around one or a few percent of the space needed for the data. Therefore we try to keep the mask encoding as simple as possible. A more complicated encoding would hardly pay off in terms of better overall compression.

If all valid pixels have the same value (zMax = zMin), or all pixels are invalid, then the encoding ends here.

Pixel Values

From here on, the valid / invalid pixels mask is known. Both in encoding and decoding. For encoding, there are 3 possible modes

- Main or block mode. Divide the image into 8x8 or 16x16 blocks (micro block size, see blob header). For each such block, determine block statistics. Based on this block statistics, compress the block. Each block can be compressed and encoded differently.

- Huffman mode. Currently only for 8 bit data (byte or char data type) lossless compression (MaxZError < 1).

- Raw binary mode, write valid data raw binary in one sweep.

The encoder should always pick the mode resulting in the best compression possible. The decoder should assume as little as possible. For instance, the decoder should work for any micro block size, even though only 8 and 16 are currently used.

We go now through these 3 modes, from easier to more complex, starting with the raw binary mode.

1. Raw Binary Mode

This mode can be called if only few pixels are valid, such as just one corner of the image. If a large raster is tiled up, then the lower right corner tile could be such a tile with only a small fraction of the pixels valid. In that case it might be best to simply write out the few valid pixels as raw binary. Another example is the lossless encoding of floating point data. The only compression, if any, is that only valid pixel values are encoded. The bit mask above has the information which pixels are valid and which are not. In the byte stream, we have a single byte flag for this mode:

Item	Format	Size
Flag for write data raw binary in one sweep (1 = yes, 0 = no)	Byte	1

If this byte is 1, all valid pixels (numValidPixels is in the Lerc blob header) are written out raw binary taking the number of bytes (numValidPixels * sizeof(data type)).

If this byte is 0, the stream continues with one of the other modes.

2. Huffman Mode

This mode is currently only tried for 8 bit input data, meaning data types byte and char, and lossless encoding, meaning MaxZError < 1. The main mode of Lerc described further below was originally designed for higher bit depth data, such as 16 bit integer or floating point data. It is not optimal for 8 bit data. For this reason this Huffman mode was added later for the important case of encoding 8 bit data lossless. If the bit depth is more than 8 bit, or if the encoding is not lossless (MaxZError >= 1), then the Huffman mode is currently not used.

The image is Huffman encoded as follows. First, the image is turned into a difference or delta image, to make use of the strong local correlation usually present in 2D images. For each pixel, if the left neighbor pixel exists and is valid, subtract its value. If that is not the case, try with the pixel above. If that is not available, subtract the last known valid pixel value. The last known valid pixel value gets initialized with 0.

Second, we compute the histogram over these delta values. When computing the deltas, we make use of the overflow properties of the byte and char data types. For example, (Byte)2 – (Byte)255 = (Byte)3. Because on decode (Byte)255 + (Byte)3 = (Byte)2. This way the size of the delta histogram is only 256, not 512. The delta histogram has usually a strong peak at or close to 0.

Third, from this histogram, we compute the optimal Huffman code book. Both histogram and Huffman code book are computed once for the entire image. From this the number of bytes needed for Huffman encoding the image can be directly calculated without actually doing the encoding. This way the encoder can compare to the other modes and decide which one is the best to use.

Item	Format	Size
Flag for Huffman encoding (1 = yes, 0 = no)	Byte	1

If the Huffman mode wins, the byte stream continues as follows.

Item	Format	Size
Huffman Format Version (= 4)	Int	4
Huffman blob size in bytes	Int	4
Min Index for non-zero codes (e.g., 217)	Int	4
Max Index for non-zero codes, exclusive (e.g., 321)	Int	4

As the delta histogram often contains a long stretch of zeros, we store the min and max indexes for which we have non-zero Huffman codes. If the histogram has entries for all indexes, then this index range is simply [0, 256]. If we have a peak around 0, then this range could be [217, 321], using wrap-around. The code for index 320 would be the code for index (320 – 256) = 64. After this header, there are three data sections:

- For the range of non-zero codes, encode the code lengths, index by index or symbol by symbol, bit stuffed at a fixed bit length which is determined by the largest code length. As the max code length allowed is hard coded as 32, the code lengths cannot take more than 6 bits per element which is hardly ever reached.

- For the same range, encode the Huffman codes. As their lengths are already encoded above, we can bit stuff them directly into an array of unsigned integer uint32.

- Encode all valid pixels of the image. For each pixel delta value, bit stuff its variable length Huffman code into an array of unsigned integer uint32. (The encoding of the pixel delta values is the same as for the Huffman codes.)

Let’s go through this in more detail. For the first section, we use a more general module for bit stuffing an array of unsigned integer into an array of bytes. The same module is also used by the main or block Lerc mode as described further below. It starts with a small header consisting of 2 numbers. The first one is a byte and contains the number of bits used to encode and bit stuff all elements of the array. As the length of the Huffman code cannot exceed 32 bits (the upper limit set in the encoder), the largest unsigned integer possible to be bit stuffed is 32. This corresponds to length 6. The first 5 bits (0-4) are used to encode this length (here, up to 6). The upper 2 bits (6-7) are used to encode the type of the second number in the header. Depending on the size of the array to be encoded, a uint (0), ushort (1), or byte (2) can be used. Here, with a max number of 256 elements, either 1 or 2 bytes will be needed to encode that number.

Item	Format	Size
Number of bits per element (bits 0-4). Bit 5 = 0 here. Bits 6-7 encode the next type.	Byte	1
Number of fixed length values	UInt (0), UShort (1), or Byte (2)	1 or 2 (or 4, rare)

After this header, the unsigned integer data are bit stuffed using the fixed bit length into a byte array.

The second section is easier to encode, we don't even need a header. We simply bit stuff the Huffman codes in the same order as the Huffman code lengths into an array of uint32. This time the bit length is variable, not fixed. As these code lengths are encoded above, the codes are easy to retrieve.

The third section is the pixel values or data. They get encoded just the same as the Huffman codes. The difference is in the decoding. For the variable length Huffman codes representing the pixel values, the lengths of these codes are not known when decoding them. Lookup tables can be used for their decoding.

Remark about encoding the Huffman codes. Since Huffman code book version 4 canonical Huffman codes are used. Meaning they could be constructed by the decoder from the code lengths alone. Despite that, we decided to keep writing and reading the Huffman codes. Reasons:

- It increases the decoder's forward compatibility. We can still change the code without breaking existing decoders.
- It reduces the decoder's complexity. Reading the code is easier than having to construct it. It is easier for a JavaScript or Python developer to write a Lerc decoder.
- The savings from not writing them are small, only a few hundred bytes.

3. Main or Block Mode

General remarks, most of which are not mandatory for the byte stream definition:

- Both the image dimensions (width and height) and the pixel is valid / invalid bit mask are known at this point, both in encoding and decoding.
- The image gets divided into square blocks of same size, with 8x8 pixels being the size most often used. This size is encoded in the Lerc header as “micro block size”. The blocks are written left to right, and row by row.
- The last block in a row or column can have different size, down to 1x1 pixels.
- For 8 bit data lossless, both block mode and Huffman mode are tried, and the mode giving better compression is taken. A byte flag is written indicating this choice.
- When the block mode is tried, the data is usually not encoded. Instead block statistics is collected to calculate the compressed block size in bytes.
- After the total number of bytes for block size 8x8 pixels is known, and if the resulting bpp <= 1, block size 16x16 pixels is also tried, to reduce the block header overhead.

Encoding one block of N x N pixels:

- Find min and max values zMin and zMax. Count number of valid pixels. Count how many consecutive pixels have the same value.
- Cover special cases: All pixels are invalid, all pixels are 0, or all pixels have the same value z0. In that case, the entire block is encoded as one byte, or as one byte plus offset z0.
- Check if the data can / should be quantized at all. Otherwise encode that block raw binary uncompressed. For instance, if the data type is float, and the value range is large while MaxZError is small (caller asks for high precision), then quantizing the data may not reduce the bpp needed to encode. Here we demand that the max unsigned integer created by the quantization is less than 2^30. This is for types of 32 bits or larger, such as int, uint, float, double. For 16 bit types, we demand max < 2^15, accordingly.
- Write the block header as small as possible. For the offset z0, we write it using a variable data type. For instance, if the data type is integer (int32), but the offset can be written as a short (int16), then it is written as a short. The byte before the offset encodes the data type used in the two upper bits 6-7. For a detailed description of the LERC block header, see below.
- Quantize and bit stuff the data. The pixel values are quantized based on the offset zMin and the error tolerance MaxZError as given by the caller. The valid pixel values get converted into an array of positive integer values (uint32). This

array is bit stuffed into a byte array using a fixed bpp based on the max value. An encoding example is given further below.

- If there are enough consecutive pixels having the same value, also try the lookup table (LUT) mode for this block. If there are only a few different (quantized) values present in a block, create an LUT from them, and assign the small index value to each pixel pointing to the correct entry in the LUT. Both the LUT and the array of indexes are then bit stuffed into a byte array using a fixed bpp based on their max values.

LERC Block Header

Item	Format	Size
Encoding Type (bits 0-1): Value = 0: raw binary Value = 1: quantized and bit stuffed Value = 2: all 0 (encode only this byte) Value = 3: all constant (encode only this header) For cases 0 and 2, the rest of the header is skipped. Bits 2-5 contain a 4 bit code for the decoder to check the integrity of the data. Bits 6-7 encode the next type.	Byte	1
Offset	Variable Type	8, 4, 2, or 1

Variable Type as used in the LERC Block Header

The idea is to use the smallest data type needed to write a data value without loss, and encode that type used in the upper 2 bits 6-7 in the byte right before the variable type entry in the block header.

The 8 bit data types byte and char don't need downsizing, which gets encoded as 0 in the byte before.

The unsigned integer types are also straightforward: ushort (0) tries byte(1). Uint (0) tries ushort (1) and byte (2).

For the signed integer types, we chose: short (0) tries byte (1) and char (2). Int (0) tries ushort (1), short (2), and byte (3).

For the floating point types float and double, we chose: float (0) tries short (1) and byte (2). Double (0) tries float (1), int (2), and short (3).

Data Type Orig / Used	Char	Byte	Short	UShort	Int	UInt	Float	Double
Char	0							
Byte		0						
Short	2	1	0					
UShort		1		0				
Int		3	2	1	0			
UInt		2		1		0		
Float		2	1				0	
Double			3		2		1	0

Array of unsigned integer (uint32) values bit stuffed using a fixed number of bits per element

Here we describe a core module. It is used to write out the quantized values for each block (in the block mode), or to write out the Huffman code lengths (in the Huffman mode). It starts with a small header:

Item	Format	Size
Number of bits per pixel (bits 0-4). Bit 5 flags simple bit stuffing (0) or LUT (1) mode. Bits 6-7 encode the next type.	Byte	1
Number of fixed length values	UInt (0), UShort (1), or Byte (2)	1 or 2 (or 4, rare)

By far the most common case is simple bit stuffing (bit 5 = 0). Here the N different values are bit stuffed into a byte array using a fixed number of bits per pixel, as determined by the largest element in the array.

If the LUT mode offers better compression (bit 5 = 1), then the different array elements are sorted in increasing order. When encoding a LERC block, the min quantized value is always 0 and is therefore omitted in the LUT. The following is written: A byte containing the full size of the LUT (including the leading 0), the LUT bit stuffed using the number of bits per pixel as specified in the above header (without the leading 0), and finally the indexes to the LUT for all valid pixels in the block, also bit stuffed at a fixed number of bits per pixel as determined by the size of the LUT. Note that this bpp must be smaller than the one for the LUT itself, otherwise the LUT mode would not have been chosen.

Built-in Integrity Checks

We added more rigorous integrity checking starting with version Lerc2 v3. After encoding the data into a Lerc memory blob, we compute the Fletcher32 checksum starting at the memory address behind the checksum field in the Lerc header and ending with the end of the blob. The result gets written to the checksum field in the header. Before decoding the Lerc memory blob, we first compute the checksum the same way as above. (We also check that the buffer we got is large enough to hold the Lerc blob based on the blob size stored in the Lerc header.) Then we compare the checksum just computed to the one stored in the header and return false if they differ. The previous consistency checks stay in place as described below.

Wherever possible, intermediate data, such as bit or byte lengths, value ranges, number of elements etc are checked for consistency. Both in the encoder and decoder. If anything inconsistent or out of order gets detected, encoder or decoder return false.

As an extra consistency check, the 4 bits 2-5 in the first byte of the Lerc blob header are filled with the simple "control signature" (15 & (j / 8)) with j being the image column coordinate of the upper left pixel of the block being encoded. The decoder checks this code and returns false if it should not match.

Example for encoding one block using MaxZError = 0.01

Let's go through all steps of encoding one 4 x 4 pixels sample data block. Note that blocks are usually larger than that, 8 x 8 or more pixels.

1234.1234	1241.8741	1256.2759	1267.2950
1280.8725	1248.2917	1272.7511	1279.3802
void	1222.2943	1239.3072	void
1264.9720	1250.0852	void	void

Step 1: Calculate the basic statistics for this data block

Min = 1222.2943

Max = 1280.8725

Number of valid pixels = 12

Number of invalid pixels = 4

Step 2: Determine how to encode this block. Assume MaxZError = 0.01.

(Max – Min) / (2 * MaxZError) = 2,928.91

This is less than 2^30. So we can quantize the pixel values from the above block, using

n(i) = (unsigned int)((x(i) – Min) / (2 * MaxZError) + 0.5)

resulting in

591	979	1699	2250
$\boxed{2929}$	1300	2523	2854
void	$\boxed{0}$	851	void
2134	1390	void	void

Step 3: Determine number of bits needed and bit stuff these non-negative integers

NumBits = ceil(log_2(2929)) = 12

Or, in other words,

2^11 < 2929 < 2^12

So we need 12 bit per number to encode all numbers of this block lossless. There are 12 valid numbers, resulting in 12 x 12 = 144 bit total. As 144 / 8 = 18, we need 18 bytes to encode the entire data block.

Step 4: Write the combined block header (Lerc block header + bit stuffed array header)

Item	Value	Size
Encoding Type (bits 0-1): Bits 6-7 encode the next type	1	1
Offset	1222.2943	4
Number of bits per pixel (bits 0-4): Bits 6-7 encode the next type	12	1
Number of valid pixel	12	1

So we need 7 bytes for the block header. The total number of bytes needed for this block is 18 + 7 = 25. Obviously the header takes too much space compared to the raw data. That’s why the smallest block size we work with is usually 8 x 8 pixels, not 4 x 4.

Example for encoding one block using MaxZError = 1.0

Let's encode the very same pixel block again, but this time using a larger error threshold of 1.0. Step 1, calculate the statistics, is the same as before.

Step 2: Determine how to encode this block. Assume MaxZError = 1.0.

(Max – Min) / (2 * MaxZError) = 29.29

This is less than 2^30. So we can quantize the pixel values from the above block, using

n(i) = (unsigned int)((x(i) – Min) / (2 * MaxZError) + 0.5)

resulting in

6	10	17	23
29	13	25	29
void	0	9	void
21	14	void	void

Step 3: Determine number of bits needed and bit stuff these non-negative integers

NumBits = ceil(log_2(29)) = 5

Or, in other words,

2^4 < 29 < 2^5

So we need 5 bit per number to encode all numbers of this block lossless. There are 12 valid numbers, resulting in 5 x 12 = 60 bit total. As 60 / 8 = 7.5, we need 8 bytes to encode the entire data block.

Step 4: Write the combined block header (Lerc block header + bit stuffed array header)

Item	Value	Size
Encoding Type (bits 0-1): Bits 6-7 encode the next type	1	1
Offset	1222.2943	4
Number of bits per pixel (bits 0-4): Bits 6-7 encode the next type	5	1
Number of valid pixel	12	1

So we need 7 bytes for the block header. The total number of bytes needed for this block is 8 + 7 = 15.