JPEG Compression

Document: JPEG Compression.doc
Version 1.0

WinZip Computing

Table of Contents

1. PREREQUISITES 3

2. OVERVIEW 3

3. ZIP ENHANCEMENTS 4

3.1 METHOD AND VERSION 4
3.2 PROPERTIES HEADER 4

4. COMPRESSED STREAM 4

4.1 BUNDLES 4
4.1.1 Headers 4
4.1.2 Metadata 5
4.1.3 Scan Data 5

5. JPEG COMPRESSION 5

5.1 VALIDATION 5
5.2 METADATA PARSING 7
5.3 METADATA COMPRESSION 7
5.4 SCAN DECODING 7
5.4.1 Scan Slice 9
5.4.2 Restart Markers 9
5.5 SCAN COMPRESSION 10
5.5.1 Entropy Encoding 10
5.6 BLOCK COMPRESSION 10
5.6.1 Notation 10
5.6.2 Primitives 10
5.6.2.1 SUM 10
5.6.2.2 AVG 11
5.6.2.3 BDR 11
5.6.3 Category 12
5.6.4 Binarization 12
5.6.5 End of Block 12
5.6.5.1 EOB Encoding 13
5.6.5.2 EOB Context 13
5.6.6 AC Coefficients 13
5.6.6.1 Zero/Non-Zero Encoding 13
5.6.6.2 Pivot Encoding 14
5.6.6.3 Value Encoding 14
5.6.6.4 Sign Encoding 15
5.6.6.4.1 Sign Context 15
5.6.7 DC Coefficient 16
5.6.7.1 DC Prediction 16
5.6.7.2 Prediction Refinement 17
5.6.7.3 DC Encoding 17
5.6.7.3.1 Value Encoding 17
5.6.7.3.2 Sign Encoding 18

6. ARITHMETIC CODER 18

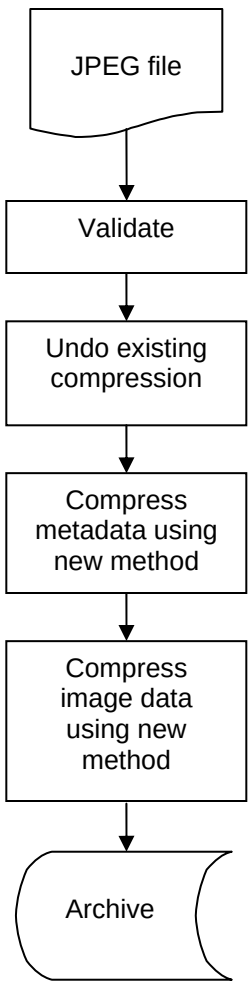
1. Prerequisites

This document assumes familiarity with the ZIP file format (Application Note 6.3 or later) together with knowledge of the JPEG image compression standard, LZMA encoding (Lempel-Ziv-Markov chain-Algorithm) and entropy coding methods.

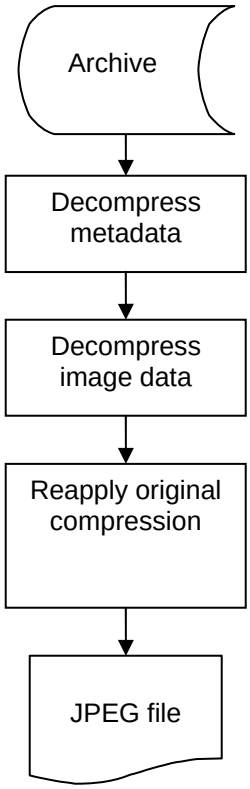
2. Overview

The JPEG compression algorithm is designed to compress image files created using the Joint Photographic Experts Group (JPEG) standard. JPEG files are inherently difficult to compress because of their built-in compression based on a combination of run-length and entropy coding techniques. The JPEG compression algorithm works by first unwinding this preexisting compression and then recompressing the file using an improved method that typically results in a 20-25% savings in space. The algorithm is lossless and reversible so that when the file is decompressed, the original entropy coding can be reapplied resulting in a bit for bit match with the original.

The following diagram illustrates the process:



Compression



Decompression

3. ZIP Enhancements

JPEG compression is integrated into the ZIP file format consistent with other compression methods described in the published Application Note on the ZIP file format.

3.1 Method and Version

The compression method field within the ZIP Local and Central Header records is set to the value 96 to indicate the data is compressed using JPEG Compression. The Version needed to extract field within the ZIP Local and Central Header records is set to 2.0; the same value as with the Deflate algorithm.

3.2 Properties Header

The JPEG compressed data stream is prefixed with a Properties Header that immediately follows the Local Header and if present, the Encryption Header. The Properties Header stores information needed to decompress the compressed data. The format of the Properties Header is as follows:

Field	Size	Description
Properties Size	1 Byte	Size of the properties header (in bytes) including this field. The minimum value is 4.
Version Information	1 Byte	JPEG compression version number. The major version number is stored in bits 4-7. The minor version number is stored in bits 0-3. Currently set to 0x10 (1.0).
Compression Method	1 Byte	JPEG compression method identifier. A value of 0 is explicitly invalid. Currently set to Method 1.
Options	1 Byte	Options specific to Compression Method. For Method 1, bits 0-4 contain what is described as a slice value. Currently set to 8. All other bits are set to 0.

Properties Header

4. Compressed Stream

The compressed data stream encompasses one or more bundles consisting of metadata (JPEG markers) followed by the image scan data. For baseline images there are usually two bundles; a larger bundle containing most of the metadata and scan data and a smaller bundle that indicates the end of the image. The bundle layout is shown here:

Primary Header	Extension Header (optional)	Metadata	Scan data (optional)

Bundle Layout

4.1 Bundles

4.1.1 Headers

The Bundle Headers consist of two fields, Uncompressed Size and Compressed Size, which represent the metadata:

Field	Size
Uncompressed Size	2/4 Bytes
Compressed Size	2/4 Bytes

Bundle Header

For the Primary Header, the width of these fields is 2 bytes (16-bits) allowing for metadata sizes up to 65534 bytes. If the metadata exceeds 65534 bytes then both fields are set to 0xFFFF and the optional Extension Header is used. The Extension Header has the same format as the Primary Header, except the fields are widened to 4 bytes (32-bits) allowing for metadata sizes up to the supported maximum of 16 MB.

4.1.2 Metadata

The JPEG metadata follows the Primary Header and if present, the Extension Header. The metadata is compressed using the LZMA compression algorithm. Details of the LZMA implementation are further described in section 5.3. All metadata up to and including the SOS (Start of Scan) or EOI (End of Image) markers is compressed as a single unit. If the compressed size exceeds the uncompressed size then the Compressed Size field of the bundle header is set to zero and the metadata is stored without compression.

4.1.3 Scan Data

The scan data follows the metadata if the SOS marker is present in the metadata. The original Huffman encoded scan data is first decoded into 8x8 DCT blocks and then recompressed using an improved entropy coder. Compression of the DCT blocks forms the core of the JPEG compression algorithm and is fully described in section 5.

5. JPEG Compression

JPEG files are normally either JPEG File Interchange Format (JFIF) files or Exchangeable Image File Format (Exif) files with the latter being used by most digital cameras. Both formats are based on the JPEG Interchange Format (JIF) as specified in Annex B of the standard. The differences between the two are small, relating to a subset of markers. The marker differences are inconsequential to the compression algorithm so both formats are readily supported.

The JPEG compression algorithm supports the following image types:

- Baseline and extended (sequential) encoding
- 8 or 12 bits/sample
- Scans with 1, 2, 3 or 4 components
- Interleaved and non-interleaved scans

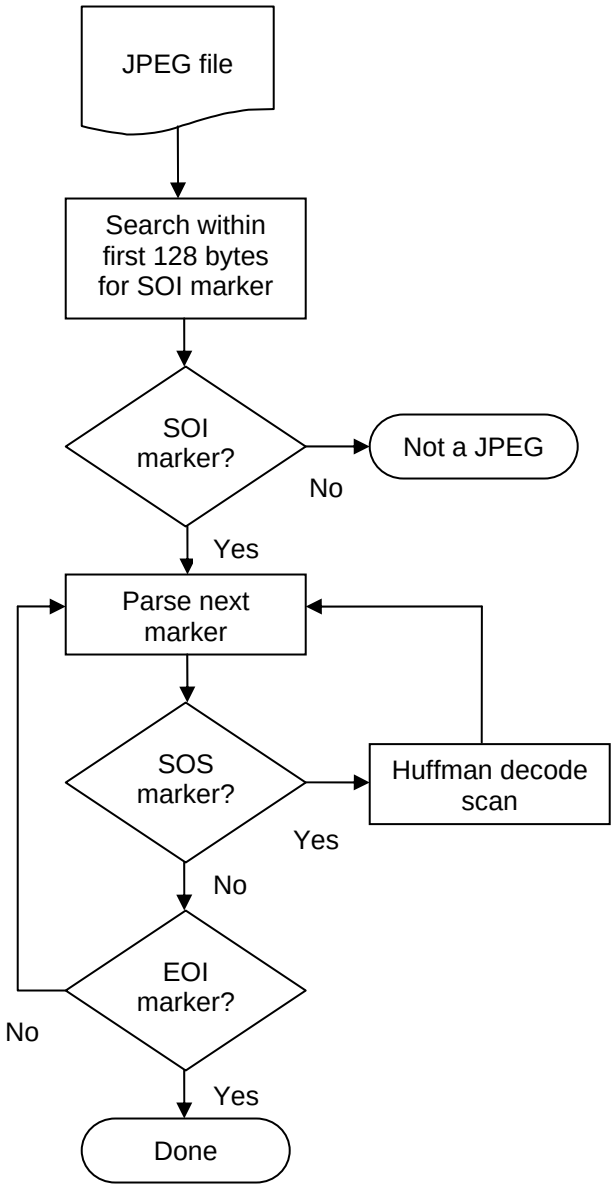
5.1 Validation

Before compressing a JPEG file its contents are validated. Validation is necessary because the JPEG algorithm requires analyzing the structure of an image, thereby requiring unsupported or corrupt JPEG images be rejected. Also, validation ensures that the original file can be reconstructed exactly when decompressed.

As part of the validation process the metadata markers are parsed and verified. There are several markers required by the compression algorithm and those not required are preserved. The primary marker that identifies a JPEG file, SOI (Start Of Image), must be found within the first 128 bytes of the file. All data preceding the SOI marker is considered unknown metadata.

To complete the validation process the image scans are decoded to ensure there are no errors with the entropy encoding. After the last scan is decoded the EOI (End Of Image) marker is parsed. Any data beyond the EOI marker is considered unknown metadata.

The following diagram illustrates the process:



Validation

5.2 Metadata Parsing

The parser recognizes the following markers. The first five markers are required by the compression algorithm:

Marker	Description	Required
SOF0, SOF1	Start Of Frame (Baseline, Extended Sequential)	✓
DHT	Define Huffman Table	✓
DQT	Define Quantization Table	✓
SOS	Start Of Scan	✓
SOI	Start Of Image	✓
EOI	End Of Image	
RSTx	Restart	
DRI	Define Restart Interval	

Recognized Markers

It is possible that the end of the image is reached without finding the EOI marker. In this case, the image is technically malformed but the situation is tolerated and handled as if the EOI marker was found.

5.3 Metadata Compression

All metadata up to and including the SOS or EOI markers is compressed as a single unit. Any data beyond the EOI marker is compressed along with the EOI marker. The data is compressed using the LZMA compression algorithm. Currently, version 4.57 of LZMA SDK is used though it is assumed newer versions will work equally well. The Coder Properties are left in their default state. The dictionary size varies from a minimum of 1k to a maximum of 512k with intermediate sizes determined using the formula:

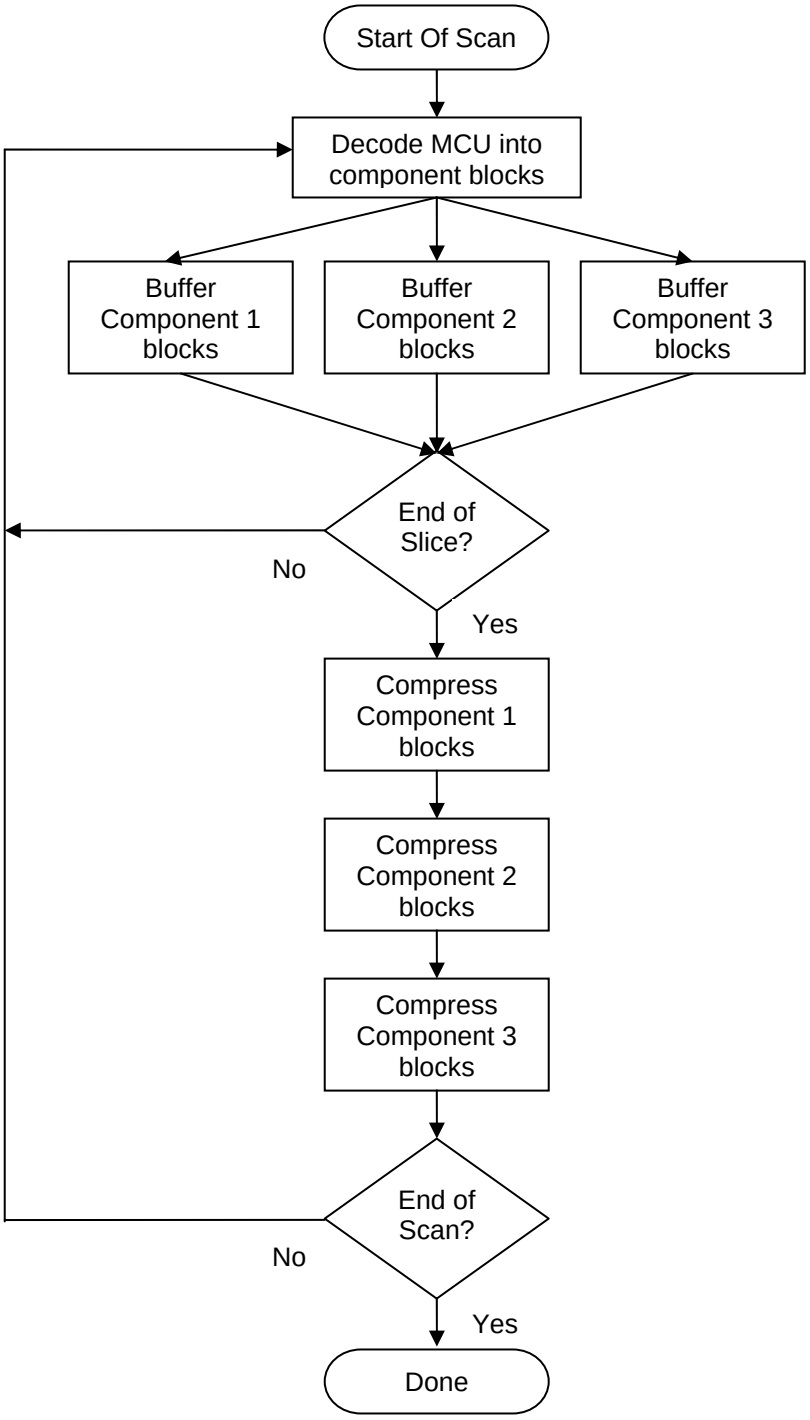
```
DictionarySize = ceil(MetadataSize / 512) * 512
```

Note that the Coder Properties are not actually written to the compressed stream; that is to say, the WriteCoderProperties() method is not invoked. Instead, the LZMA compression parameters are inferred from the Properties Header as described in section 3.2.

Prior to writing the LZMA compressed stream a Bundle Header is initialized and written as described in section 4.1.1.

5.4 Scan Decoding

The SOS marker initiates the start of the entropy coded scan data. For interleaved scans the MCU component blocks are decoded and separated, as the compression algorithm works with each component independently. If there are more than a given number MCUs within a scan (characterized as a slice) then the scan is sectioned into slices in an effort to limit the amount of buffering required. After the DCT blocks have been decoded they are passed one by one to the improved compression engine. This process continues until the end of the scan is reached as shown in the following diagram:



Scan Decoding

5.4.1 Scan Slice

Slices are a mechanism to reduce the amount of buffering required when decoding a scan. They are optional and the compressor determines their use. The size of a slice is variable. A slice size of 0 indicates slices are not used; otherwise, the slice size is specified as a 2^n count of MCUs. The minimum slice size is 2^7 and the maximum size is 2^{37} — represented as values in the range of 1 to 31 respectively. The following table shows all possible slice values and the equivalent 2^n sizes:

Value	2^n Size	Value	2^n Size
0	—	16	2^{22}
1	2^7	17	2^{23}
2	2^8	18	2^{24}
3	2^9	19	2^{25}
4	2^{10}	20	2^{26}
5	2^{11}	21	2^{27}
6	2^{12}	22	2^{28}
7	2^{13}	23	2^{29}
8	2^{14}	24	2^{30}
9	2^{15}	25	2^{31}
10	2^{16}	26	2^{32}
11	2^{17}	27	2^{33}
12	2^{18}	28	2^{34}
13	2^{19}	29	2^{35}
14	2^{20}	30	2^{36}
15	2^{21}	31	2^{37}

Slice Values

The slice value is stored in the Options field of the Properties Header as described in section 3.2. The slice size is determined by converting the 2^n size to a multiple of the row size for the scan. Using *X* and *Y* to denote the scan's dimensions in MCUs, the slice size is determined using the following formula:

$$SliceSize = \text{ceil}(Y \,/\, \text{ceil}(Y \,/\, \max(2^nSize \,/\, X, 1))) * X$$

At the end of each slice, the encoder's FLUSH routine is called but its state otherwise remains unchanged.

5.4.2 Restart Markers

Restart Markers are recognized within the entropy coded data but are not explicitly written to the compressed data stream. During decompression when the entropy coding is reapplied, the markers are recreated at the defined restart interval.

5.5 Scan Compression

For interleaved scans, compression is performed on a component by component basis in the same order as encountered within the MCU. Non-interleaved scans have a single component by definition. Each of the component's 8x8 DCT blocks are Huffman and run-length decoded. The DCT coefficients remain quantized and in zigzag scanning order. In addition to the current block, two additional DCT blocks must be maintained for the compression engine; the block above and the block to the left of the current block. These are referred to as the North and West blocks respectively.

5.5.1 Entropy Encoding

Component blocks are compressed using binary arithmetic coding (BAC) combined with unique probability models that form the core of the compression algorithm. When working with the combination of interleaved scans and slices, there is a requirement that concurrent BAC states are kept for each component in the scan. The reason for this is that states are preserved across slice boundaries.

5.6 Block Compression

The component's quantization table, North and West blocks and previously seen coefficients within current block encompass all the requirements of the block compressor. The EOB is compressed first followed by the AC terms in the order EOB to AC1 followed by DC.

5.6.1 Notation

We use B to denote an 8x8 DCT block, B_c to denote the current block, B_n to denote the neighboring block North of B_c and B_w to denote the neighboring block West of B_c. We use k to denote the k^{th} zigzag position within a block and $B[k]$ to denote the DCT coefficient at the k^{th} position of B. Where the component's quantization table is required, S denotes the quantization table in zigzag order.

5.6.2 Primitives

There are three primitives that combined, form the basis for block compression. They are identified as SUM, AVG and BDR and are functions of k and/or B_c, B_w, and B_n. Along the edge of a scan where B_w or B_n may not exist, they are substituted by a null block – i.e., a block where all coefficients are zero.

5.6.2.1 SUM

SUM(B, k) is defined as the sum of all coefficient absolute values below and to the right of k within a block. The x's in the diagram below illustrate the case of $k = 24$:

			k	*x*	*x*	*x*	*x*
			x	*x*	*x*	*x*	*x*
			x	*x*	*x*	*x*	*x*
			x	*x*	*x*	*x*	*x*
			x	*x*	*x*	*x*	*x*

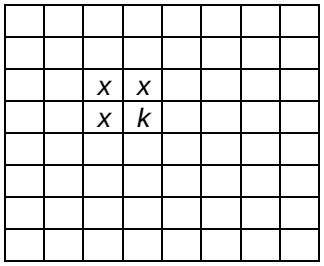
8x8 DCT Block

Each x is summed using:

```
sum += abs(B[x])
```

5.6.2.2 AVG

AVG(B_n, B_w, k) is defined as the average of coefficient absolute values at k and the coefficient absolute values at positions directly above, to the left and to the upper-left of k for both the North and West blocks. The x's in the diagram below illustrate the case of $k = 24$:



8x8 DCT Block

If k is located near the top or left border of a block, only the available x's participate. AVG also considers coefficient quantization. Each x is summed using:

```
sum += (abs(Bn[x]) + abs(Bw[k])) * S[x] / S[k]
```

The coefficient at k is then added and the average is computed:

```
avg = (sum + abs(Bn[k]) + abs(Bw[k]) + count) / (2 * count)
```

5.6.2.3 BDR

BDR(B_c, B_n, B_w k), is valid when k is located within the top or left border of a block. The coefficient at k and the coefficient at positions directly below or to the right of k participate. The x's in the diagrams below illustrate all possible k:

	0	1	2
0			
1	k	x	
2	k	x	
3	k	x	
4	k	x	
5	k	x	
6	k	x	
7	k	x	

West Block

	0	1	2	3	4	5	6	7
0		k	k	k	k	k	k	k
1		x	x	x	x	x	x	x
2								

North Block

BDR also considers coefficient quantization. For k within the first row, the computation is:

```
bdr = Bn[k] – (Bn[x] + Bc[x]) * S[x] / S[k]
```

For k within the first column, the computation is:

```
bdr = Bw[k] – (Bw[x] + Bc[x]) * S[x] / S[k]
```


5.6.3 Category

The concept of a value's magnitude, designated its category, is used extensively by the compression procedures. Mathematically it's expressed as:

$$cat = \text{ceil}(\log_2(\text{abs}(value) + 1))$$

A sample of values and the equivalent categories is shown here:

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

Categories

The function CAT(*N*) is defined as returning the category of any positive integer *N*.

5.6.4 Binarization

The coding of coefficient values is done using a scheme called binarization. Binarization essentially maps a positive integer into a universal code. A generalization of Elias gamma coding is used. The table below shows how the code compares with other codes:

Value	Implemented Code	Elias Gamma	Exp-Golomb (k=0)
0	0	—	0
1	10	0	10:0
2	110:0	10:0	10:1
3	110:1	10:1	110:00
4	1110:00	110:00	110:01
5	1110:01	110:01	110:10
6	1110:10	110:10	110:11
7	1110:11	110:11	1110:000
8	11110:000	1110:000	1110:001
9	11110:001	1110:001	1110:010

Code Comparison

Codes are mapped across multiple BAC bins based on context. Generally, the unary part of the code maps to one set of bins and the binary remainder another set.

5.6.5 End of Block

The first step prior to coding the AC coefficients is to determine the End of Block (EOB). The EOB is defined as the last non-zero coefficient for the block in zigzag scanning order. Values within the range of 1 to 63 coincide with terms AC1 to AC63 respectively. The value 0 indicates that all AC terms are zero.

5.6.5.1 EOB Encoding

Encoding of EOB is accomplished using a binary tree. Six bits are needed to represent all possible EOB values, resulting in 63 bins allocated to the tree as follows:

Bin	Bit
0	5
1-2	4,4
3-6	3,3,3,3
7-14	2,2,2,2,2,2,2,2
15-30	1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
31-62	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

Bin Allocation

5.6.5.2 EOB Context

The context used to encode EOB is determined from the North and West blocks. SUM(B, 0) is evaluated for the neighboring blocks and the average is computed:

$$avg = (\text{SUM}(B_n, 0) + \text{SUM}(B_w, 0) + 1) / 2$$

Along the edges of the scan only a single block participates:

$$avg = \text{SUM}(B_n, 0)$$

or

$$avg = \text{SUM}(B_w, 0)$$

For the very first block where no neighbor exists, the context is 0. For all other blocks, the context is determined by computing:

$$ctx = \min(\text{CAT}(avg), 12)$$

The total number of bins required to encode EOB is therefore 63∗13 or 819.

5.6.6 AC Coefficients

If EOB is zero then the coding of AC coefficients is not necessary. Coding proceeds with the coding of DC. For a non-zero EOB, AC coefficients are coded in zigzag order from highest to lowest (EOB to AC1). For all coefficients other than at EOB, a zero/non-zero decision is coded, a pivot, an absolute value and a sign. By definition, the coefficient at EOB is a non-zero value, thus the zero/non-zero decision is not necessary.

5.6.6.1 Zero/Non-Zero Encoding

Zero/Non-Zero codes the decision ($B[k] \neq 0$). Encoding of zero/non-zero is not required for the coefficient at EOB.

For k located within the first row or column of the block, BDR(B_c, B_n, B_w, k) is evaluated:

$$val_1 = \text{abs}(\text{BDR}(B_c, B_n, B_w, k))$$

For all other k, AVG(B_c, k) is evaluated:

$$val_1 = \text{AVG}(B_c, k)$$

The SUM(B_c, k) is then evaluated:

val_2 = SUM(B_c, k)

The context to encode zero/non-zero is then determined by computing:

ctx = ((k-1) * 3 + min(CAT(val_1), 2)) * 6 + min(CAT(val_2), 5)

The total number of bins required is therefore 62∗6∗3 or 1116.

5.6.6.2 Pivot Encoding

Pivot codes the decision (abs(B[k]) ≥ 2). Encoding of pivot is not required if the AC value is zero.

For k located within the first row or column of the block, BDR(B_c, B_n, B_w, k) is evaluated:

val_1 = abs(BDR(B_c, B_n, B_w, k))

For all other k, AVG(B_c, k) is evaluated:

val_1 = AVG(B_c, k)

The SUM(B_c, k) is then evaluated:

val_2 = SUM(B_c, k)

The context to encode the pivot is then determined by computing:

ctx = ((k-1) * 5 + min(CAT(val_1), 4)) * 7 + min(CAT(val_2), 6)

The total number of bins required is therefore 63∗5∗7 or 2205.

5.6.6.3 Value Encoding

Encoding of AC value is not required if its absolute value is less than two. Allowable AC values fall within the range of -16383 to 16383. The sign bit is coded separately; therefore 14 bits are required to encode the value.

For k located within the first row or column of the block, BDR(B_c, B_n, B_w, k) is evaluated:

val_1 = abs(BDR(B_c, B_n, B_w, k))

For all other k, AVG(B_c, k) is evaluated:

val_1 = AVG(B_c, k)

The SUM(B_c, k) is also evaluated:

val_2 = SUM(B_c, k)

For k located within the first row of the block, *k's* column is determined. The column is adjusted to a value between 0 and 6:

val_3 = col – 1

For k located within the first column of the block, *k's* row is determined. The row is adjusted to a value between 0 and 6:

val_3 = row – 1

For all other *k*, the category of *k* is determined, resulting in a value between 0 and 6:

$val_3 = \text{CAT}(k - 4)$

In addition, for the conditions of *k* located within the first row, *k* located within the first column, and all other *k*; we shell use *n* to denote this, where $0 \leq n < 3$.

The AC absolute value is decremented by 2 and binarized as described in section 5.6.4. Coding of the unary magnitude is capped at 9. The context to encode the magnitude is determined by computing:

$ctx_m = (n * 9 + \min(\text{CAT}(val_1), 8)) * 9 + \min(\text{CAT}(val_2), 8)$

The context to encode the binary remainder is determined by computing:

$ctx_r = n * 7 + val_3$

The total number of bins required to encode the value is therefore $3*(9*9*9+13*7)$ or 2460.

5.6.6.4 Sign Encoding

Encoding of AC sign is not required if the AC value is zero. The method of encoding AC sign is determined from the location of *k* within the block. For *k* located within the first and second rows or first and second columns, as illustrated by the *x*'s in the diagram below, a context is used for the encoding of AC sign:

	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
x	x						
x	x						
x	x						
x	x						
x	x						
x	x						

8x8 DCT Block

For all other *k*, AC sign is coded with a fixed probability.

5.6.6.4.1 Sign Context

For *k* located within the first row or column of the block, BDR(*B*, B_n, B_w, *k*) is evaluated. If the result is zero, the AC sign is coded with fixed probability. For a non-zero result, the sign is taken:

$sgn = \text{BDR}(B_n, B_w, k) < 0$

For *k* located within the second row of the block with the exception of $k = \text{AC4}$, the coefficient at the same position from the North block is examined. Along the top edge of the scan where B_n does not exist, it is substituted by a null block. If the examined value is zero, the AC sign is coded with fixed probability. For non-zero values, the sign is taken:

$sgn = B_n[k] < 0$

For *k* located within the second column of the block with the exception of $k = \text{AC4}$, the coefficient at the same position from the West block is examined. Along the left edge of the scan where B_w does not exist, it is substituted by a null block. If the examined value is zero, the AC sign is coded with fixed probability. For non-zero values, the sign is taken:


```
sgn = Bw[k] < 0
```

For k = AC4, the average $B_n[k]$ and $B_w[k]$ is computed. Along the edge of a scan where B_w or B_n may not exist, they are substituted by a null block. If the computed result is zero, the AC sign is coded with fixed probability. For a non-zero result, the sign is taken:

```
sgn1 = (Bn[k] < 0) ? -1 : (Bn[k] > 0) ? 1 : 0
sgn2 = (Bw[k] < 0) ? -1 : (Bw[k] > 0) ? 1 : 0

if ((sgn1 + sgn2) != 0)
    sgn = (sgn1 + sgn2) < 0
```

There are total of 27 k for which a context is used for encoding AC sign. Using n to denote the n^{th} k for which a context is used, where $0 \le n < 27$, the context to encode AC sign is determined by computing:

```
ctx = (n * 3 + min(CAT(abs(Bc[k])) / 2, 2)) * 2 + sgn
```

The total number of bins required to encode AC sign is therefore 27∗3∗2 or 162.

5.6.7 DC Coefficient

Coding of DC follows the coding of AC coefficients. DC is first predicted using information from neighboring blocks as well as AC coefficients from the current block. The absolute value of the residual is then coded followed by the residual's sign.

5.6.7.1 DC Prediction

The initial DC prediction is based on DC values from the North and West blocks as well as AC values directly below and to the right of DC, that include the current block. The *x*'s in the diagram below illustrate:

	0	1	2	3
0	*x*	*x*		
1	*x*			
2				
3				

For the North block we calculate:

```
p0 = round(Bn[0] – 2.2076 * S[2] / (2 * S[0]) * (Bn[2] – Bc[2]))
```

And for the West block:

```
p1 = round(Bw[0] – 2.2076 * S[1] / (2 * S[0]) * (Bw[1] – Bc[1]))
```

Avoiding floating point, the computations are:

```
t0 = Bn[0] * 10000 – 11038 * S[2] * (Bn[2] – Bc[2]) / S[0]
p0 = ((t0 < 0) ? (t0 - 5000) : (t0 + 5000)) / 10000
```

And:

```
t1 = Bw[0] * 10000 – 11038 * S[1] * (Bw[1] – Bc[1]) / S[0]
p1 = ((t1 < 0) ? (t1 - 5000) : (t1 + 5000)) / 10000
```


For the very first block where no neighbor exists, the predicted DC is 0. Along the edges of a scan where only a single neighbor exists, the predicted DC is either p_0 or p_1, depending on which edge the block is located. For all other blocks, the prediction is further refined.

5.6.7.2 Prediction Refinement

Refinement of the predicted DC again uses the North and West blocks as well as the current block, using AC coefficients located within the first column and first row of the block as shown by the *x*'s below:

	0	1	2
0			
1	*x*		
2	*x*		
3	*x*		
4	*x*		
5	*x*		
6	*x*		
7	*x*		

North Block

	0	1	2	3	4	5	6	7
0		*x*	*x*	*x*	*x*	*x*	*x*	*x*
1								
2								

West Block

For the North block, we compute for each *x*:

d_0 += abs(abs(B_n[*x*]) – abs(B_c[*x*]))

And for the West block:

d_1 += abs(abs(B_w[*x*]) – abs(B_c[*x*]))

The final calculation giving our prediction is:

$DC_p = (2^{-d0} * p_0 + 2^{-d1} * p_1) / (2^{-d0} + 2^{-d1})$

Again, avoiding floating point, we compute:

```
if (d0 > d1)
    DCp = ((1 << min(d0 – d1, 31)) * p1 + p0) / (1 + (1 << min(d0 – d1, 31)))
else
    DCp = ((1 << min(d1 – d0, 31)) * p0 + p1) / (1 + (1 << min(d1 – d0, 31)))
```

5.6.7.3 DC Encoding

DC is coded using the residual from the predicted value:

$residual = B_c[0] - DC_p$

Allowable DC values fall within the range of -16384 to 16383, causing the residual to fall between -32767 and 32767. The sign bit is coded separately; therefore 15 bits are required to encode the value.

5.6.7.3.1 Value Encoding

The context used to encode the residual value is determined from the AC Coefficients. SUM(B_c, 0) is first evaluated:

sum = SUM(B_c, 0)

The context is then determined by computing:

$ctx = \min(\mathrm{CAT}(sum), 12)$

The residual absolute value is binarized as described in section 5.6.4. Coding of the unary magnitude is capped at 10. Both the magnitude and binary remainder are encoded using *ctx*.

The total number of bins required to encode the value is therefore 10∗13+14∗13 or 312.

5.6.7.3.2 Sign Encoding

Encoding of the residual sign is not required if the residual is zero. The context used to encode the sign is determined from the sign of the predicted DC and the signs of the DC values of the North and West blocks:

$sgn_n = B_n[0] < 0$
$sgn_w = B_w[0] < 0$
$sgn_p = DC_p < 0$

Along the edges of the scan, where a neighboring block may not exist, sgn_n or sgn_w is set to zero, depending on which edge the block is located.

The context is then determined by computing:

$ctx = (sgn_n * 2 + sgn_w) * 2 + sgn_p$

The total number of bins required to encode the sign is therefore 2∗2∗2 or 8.

6. Arithmetic Coder

The arithmetic coder is a high speed, finite precision, binary arithmetic coder (BAC) with probabilities represented in the logarithmic domain. Details of the coder can be found in U.S. patent no. 4,791,403. The probability table is constructed using the parameters (*kavg* = 5, *kmax* = 11) and is reproduced here:

i	*logp*	*lqp*	*nmaxlp*	*halfi*	*dbli*
0	1024	0	16384	8	0
1	895	272	16110	8	1
2	795	502	15105	7	2
3	706	726	14826	7	3
4	628	941	14444	7	4
5	559	1150	13975	6	5
6	493	1371	13804	6	6
7	437	1578	13547	6	7

8	379	1819	13265	6	8
9	331	2044	13240	6	7
10	287	2278	12915	6	7
11	247	2521	12844	6	6
12	212	2765	12720	6	6
13	186	2971	12648	6	6
14	158	3227	12482	6	6
15	143	3382	12441	6	6
16	127	3566	12319	6	6
17	110	3788	12320	6	6
18	98	3965	12250	6	6
19	84	4200	12180	8	6
20	72	4435	12168	9	6
21	65	4590	12155	10	6
22	59	4737	12154	10	6
23	53	4899	12084	10	6
24	48	5050	12096	10	6
25	45	5147	12105	10	7
26	42	5250	12096	10	8
27	40	5325	12080	9	8
28	37	5441	12062	9	9
29	35	5527	12075	8	9
30	33	5617	12078	8	10
31	30	5758	12060	7	10
32	28	5863	12068	7	10
33	26	5976	12090	6	10
34	23	6157	12075	6	10
35	21	6295	12075	6	10
36	19	6447	12103	5	9

37	17	6616	12121	5	8
38	15	6806	12150	4	7
39	13	7024	12181	4	6
40	11	7278	12221	3	6
41	9	7585	12294	3	5
42	7	7972	12411	3	4
43	5	8495	12615	3	3
44	4	8884	13120	2	3
45	3	9309	13113	2	3
46	2	10065	14574	1	2
47	1	11689	21860	0	1
48	1024	0	0	0	0

In order to support fixed statistics an additional entry is appended to the probability table (index 48). The QSMALLER and QBIGGER routines are also modified as shown in the following figures:

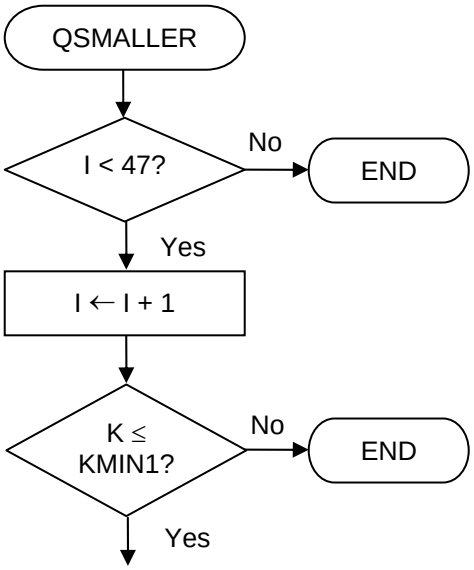


FIG. 13

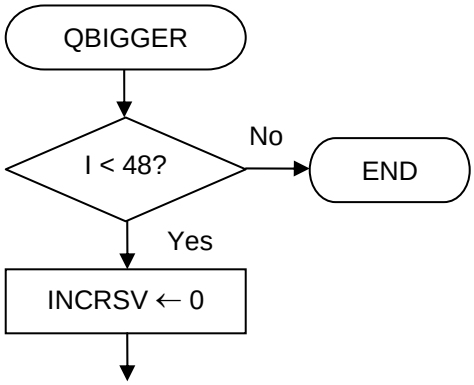


FIG. 18