JPEG/JFIF Concepts and Format

by Simon Southwell
1st July 2010

Introduction

JPEG format files are now ubiquitous on the Internet, in digital cameras and on phones, as well as in many other applications. It is no longer cutting edge technology but, because of its proliferation, is still relevant today and an understanding of the format and the coding and decoding processes can serve as a jumping off point to more advanced techniques.

In this article a discussion of the concepts of the JPEG [1] (and JFIF [2]) formats are presented in the context of going on to describe decoder software and hardware implementation. As such it is limited to those aspects of JPEG relevant to this end, rather than a definitive look at the entire standard. The article consists of three parts. The first introduces the basic concepts of JPEG, and narrows this down to details relevant to implementing a JPEG decoder. This article is meant to be read *before* tackling the implementation articles, unless familiarity of the format is already mastered. The second article details the internal architecture and design of both a software and hardware implementation of a fully working JPEG decoder, with accompanying source code available for download. The last article describes the process of development, the verification environment and the methods used for quality assurance of the IP, including simulation, bit matching, FPGA synthesis, and real-time hardware verification on an FPGA devlopment PCB platform.

Concepts

In this section is a brief introduction of the main concepts in JPEG encoding and decoding, and this will lay the context for the rest of the discussions. It is not meant to be a definitive tutorial, and more information can be found via the References section, at the end.

The main steps in encoding an image into JPEG are as follows.

- Divide the image into 8 x 8 squares.
- Transform the square into an 8 x 8 frequency oriented square using the "Discrete Cosine Transform"
- Quantise the DCT square by diving elements with a given pattern to reduce the information content
- Serialise the square's data into a byte stream
- Encode the stream with a combination of Huffman Encoding and Run Length Encoding
- Bundle the whole thing up with format information, including the tables used for quantisation and Huffman encoding.

The above simplified encoding process is just the reverse in decoding, and it does skip over some details and variations in the formatting and processing. It assumes only one colour (greyscale) and glosses over the formatting data complexities. In reality many images are colour, and the RGB data must be transformed into "luminance and chrominance" components and these transformed separately, but perhaps differently from each other. The format data itself can vary wildly between encoded files and the JFIF format is an additional complication which adds complexity over the basic format. Nonetheless, we will cover these topics in the descriptions below.

Luminance and Chrominance

As mentioned above, before we can encode our image (if it is colour), we must transform the image from RGB to luminance and chrominance. There are a number of such formats, but we will stick with one known as YCbCr. Effectively we take three components (R G B) and transform them into three new components (Y Cb and Cr). Why? Well, the Y component is the greyscale luminance-i.e. the black and white image- whilst the two other components are the chrominance, which are colour difference components. By separating the luminance and chrominance we can treat them differently, and apply compression differently, discarding information in one to a lesser or greater degree than another. Human perception is such that subtle changes in shading are perceived more keenly than subtle changes in colouring. Therefore it is possible to reduce the colour information more aggressively than the shading information whilst maintaining the integrity of the image. The format allows separate quantisation tables to be used for the luminance and chrominance components, and the chrominance quantisation will usually discard more information than for the luminance. Of course, if our image is already grey scale, then effectively we simply have the Y component already encoded, with no chrominance components.

The actual transformations themselves are a simple linear relationship:

RGB to YCbCr
Y = 0.299 R + 0.587 G + 0.114 B
Cb = -0.1687 R - 0.3313 G + 0.5 B + 128
Cr = 0.5 R - 0.4187 G - 0.0813 B + 128
YCbCr to RGB
R = Y + 1.402 (Cr-128)
G = Y - 0.34414 (Cb-128) - 0.71414 (Cr-128)
B = Y + 1.772 (Cb-128)

Table 1: Colour Space Transforms

So, using these transformations, cookbook style, it is easy to swap between the colour spaces. If floating point arithmetic is a heavy penalty (in a hardware implementation, say), then simply scaling the values and using integer arithmetic will do the trick. Having said all this, the author has come across JPEG files which are encoded directly with RGB components, and have no colour space conversion at all.

Sub-sampling

We have mentioned above how we separate the luminance and chrominance components so that we can treat them differently in terms of discarding data. Some of that comes from the quantisation of the data (see below), but also we might assume that the colour components do not change too much between adjacent 8x8 regions, or at least do not need to. So we can choose to average out the chrominance data over a larger region, and use the same data for multiple luminance components. This is called sub-sampling.

There are various sub-sampling strategies, with esoteric names (4:4:4, 4:2:2, 4:2:0 and others). However, I will only mention two common ones, and you'll get the idea from these. (The 4:4:4, by the way, is no sub-sampling-which is the default).

The first sub-sampling method is 4:2:2. Here, the chroma components of 2 adjacent horizontal 8x8 arrays are averaged. In the encoding process the two luminance arrays are transmitted (or stored) before the averaged Cb and Cr chroma arrays, which follow. On decoding, the chroma component is used for reconstruction of both 8x8 regions.

For a 4:2:0 sub-sampled image, a quartet of adjacent arrays are grouped-two horizontally, and the two immediately below them vertically. The chroma components are averaged for all four arrays, and the 4 luminance arrays transmitted before being followed by the averaged Cb and Cr arrays. It is possible that an image may be sub-sampled vertically, without horizontally as well, but this is rarely used.

Sub-sampling is the first real discarding of data in the JPEG process. It makes some assumptions about colour components, and can cause artefacts at the transitions of colour areas. But it does start the compression process off by reducing the amount of data stored or transmitted. The use of sub-sampling is part of the trade-off for compression ratios versus acceptable image reproduction. Higher quality settings (with lower compression) are unlikely to use sub-sampling, as for lower and lower settings, one is likely to use first 4:2:2 then 4:2:0 sub-sampling strategies.

DCT and iDCT

Having separated the luminance and chrominance components (if any), these can now be treated as completely separate, and the encoding process applied independently-and each process using the same steps. The 8x8 squares of data that the image has been carved into are a spacial representation of the image. It should be noted at this point that if the source image does not divide exactly into multiples of 8x8 squares, then the edge squares (right hand edge and/or lower edge) have the overspill elements set to zero. The X and Y dimensions will be stored with the image, and so these extra bits can be discarded on decoding.

The first step is to transform this data into a *frequency* representation. In terms of an image, this means the rate of change across the spatial domain-so if the image is a checker board pattern of white then black pixels, changing every pixel, then this is high frequency. If it is gradually going from white to black (say) across the 8x8 square, then this is low frequency, whilst a flat solid value is "DC".

This process of transforming the spatial data to frequency data does not compress the data whatsoever, and we turn 64 values into another 64 values-so why bother? Well, it turns out that the high frequency components are not as important in human perception as the the lower frequency ones, and the DC component is the most important of all, and we can discard some of the higher frequency data without significantly affecting the perception of the image. By transforming the image from spatial to frequency values, it allows us to manipulate the higher frequency data easily, without affecting the important low frequency components. It is hard to imagine doing this directly on the spatial data. The question now becomes; how are we to transform the data into frequency components? We could simply do an FFT transformation, and this would allow us to achieve the desired effect, but this yields complex (imaginary) values which are more compute intensive to manipulate. The JPEG committee opted for the "Discrete Cosine Transform" (DCT), which is a cousin of the FFT. The main characteristics that make it useful for our purposes are that it yields only real values (i.e. cosine components) and also orders the data within the 8x8 matrix with the DC component at (0,0), the top left corner, and the frequencies get higher as one traverses diagonally down to the lower right hand corner (see Figure 1 below). If (as discussed later) we start throwing away higher order values we may, hopefully, get zeros as we move towards the highest frequency points. Serialising the data in a special way (see quantising section later) will yield, in these cases, a run of zeros which we can use run-length encoding to compress.

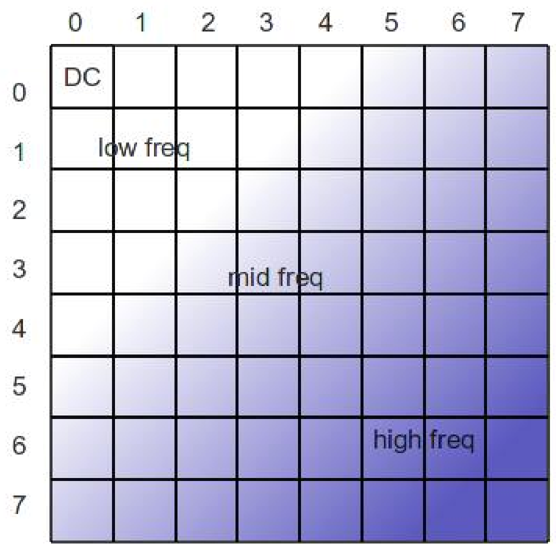

Figure 1: DCT frequency distribution of 8x8 matrix

Conceptually, the DCT manages to yield a result with only real (i.e. cosine) components by reflecting the spatial data. Imagine an N point set of data from 0 to N-1. Take the points of N-2 down to 0 and make them points N to 2N-1. As the signal is now symmetrical, when the DFT is taken, we end up with a symmetrical Fourier Transform with no imaginary components. Throw the top components away (they are a reflection of the lower half anyway) and we have the N data points transformed into N DFT values. This, as described, is for a one dimensional set of data, but it works just as well for two dimensional data. First the rows (or columns-it doesn't matter which) are transformed as a set of separate 1d transforms, and then the columns (or rows) of the resultant data are transformed in the same way. The equations for a two dimensional DFT and inverse DFT are shown in Figure 2 below.

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i)C(j) \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} pixel(x,y) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

$$pixel(x,y) = \frac{1}{\sqrt{2N}} \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} C(i)C(j)\, DCT(i,j) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

$$C(x) = \frac{1}{\sqrt{2}} \quad if\ x = 0,\ else \quad 1 \quad if\ x > 0$$

Figure 2: DCT/iDCT Formulae

These equations might look slightly intimidating at first, but a couple of things make it a lot simpler. Firstly, for JPEG, N is fixed at 8, so the 2N and 1 over root 2N all simplify to mere numbers. Even the C(i)C(j) values are either 1, 0.5 or 1/root(2). As alluded to above, an implementation need not combine the calculation of a value in a single step, but implement a 1 dimensional function, and do a two stage operation. Actually, even greater optimisations and simplifications are possible, and these will be described in the implementation sections in the articles to follow.

There is one more step taken in JPEG that is not really part of DCT transformation, but is done directly on the DCT values. The DC component, in the top left hand corner, is turned into a difference value. I.e. the value at location (0,0) is a running difference of all the previous values minus the current DCT DC value. This is done separately for each channel-i.e. the luminance (Y), blue chrominance (Cb) and red chrominance (Cr) channels each have their own running DC difference value. Within the format, means are provided for resetting the difference values. This is useful if the DC value radically changes (e.g. from light red region to dark blue region) for a large section of the image. How one goes about deciding when this is useful, and how to implement it, is beyond the scope of this article.

Quantising

Quantising is at the first real step towards compression of image data (chroma sub-sampling not withstanding-see above). As we saw before, the frequency components become less and less important in the images perception as we increase in frequency. When we have transformed the image 8x8 arrays using the DCT, we have an array with these frequencies ready for us to make adjustments. If the higher frequency components are less important, we could store them with fewer bits, and live with the errors that this introduces. To do this we can divide the actual DCT values by scaling factors from 1 to 255, where 1 makes no change, and 255 almost reduces every value to 0. The values in between allow us to weight the importance of the data across the frequency spectrum. In addition we can choose different weightings depending on some user chosen criteria. So, for instance, a high quality setting will use low value divisors, discarding little data, whilst a low quality setting uses greater divisors, and discards more of the data.

The divisor values are organised into "Quantisation tables". Some typical quantisation tables are shown in Figure 3 below:

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Figure 3: Example Quantisation Tables

The above two examples are lifted straight from the JPEG specification [1], with the left intended for luminance data, and the right intended for chrominance data. As you can see, the numbers *tend* to get larger as one traverses from the top left corner to the bottom right. Also, we have separate tables for luminance and chrominance data, allowing different compression of the two types of information, where chrominance data is more tolerant of error than luminance, as we saw earlier.

When encoding the image, these tables are used to divide the DCT amplitude values, before further encoding steps. Obviously, during decoding, we will need to multiply the data again by these values in order to reproduce the DCT-but with errors in the lower bits dependant on the size of the divisor. Therefore, the quantisation tables used must be stored or transmitted along with the picture data in order for this step to happen, and the format makes provision for this.

So, having divided the DCT data by the quantisation tables, the resultant 8x8 matrices will contain much smaller numbers. The hope is that many of these numbers are very small or even 0, and that the most likely place for this is towards the higher frequency. The smaller numbers require fewer bits to represent, and if we get a run of zeros, we can use run-length encoding to efficiently represent these values. Firstly, though, we must turn the 8x8 matrices into a stream of bytes suitable for storing or transmitting. We want to do this in such a way as to maximise the potential for run-length encoding of zeros. Figure 4 below shows the order in which we serialise the two dimensional data.

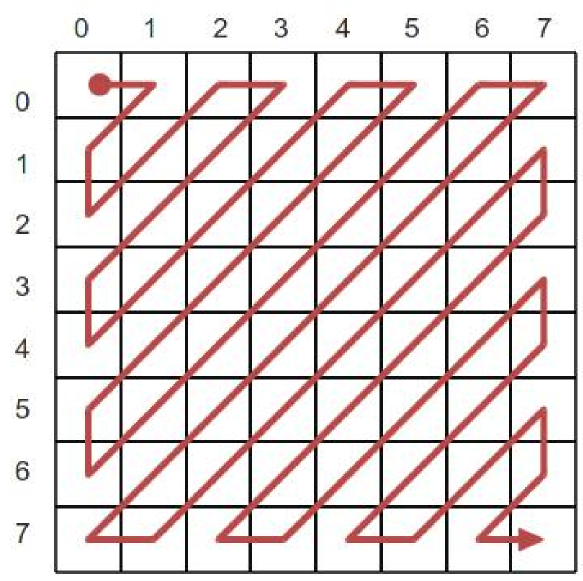

Figure 4: DCT zig-zag serialisation

As can be seen, we start in the top left corner (at DC) and work our way towards the bottom right, at the highest frequency. This pattern reflects the increase of frequency seen in Figure 1, and the values tend to be serialised from low to high frequency. Given that the quantisation step is likely to have produced more zeros towards the high frequency end, then zero values are more likely to be adjacent in the serialisation, and by putting these towards the end, then if all remaining elements are zero, we can terminate the stream, leaving them as implied, and save bits for these remaining values. Before we can serialise these quantised values, we must first reduce the bits required to store them, and this is discussed in the next section.

Variable length and Run length Encoding

The quantised values, now much smaller, require fewer bits to represent them, but we must first make this transformation. The table below shows how this transformation is done.

SSSS	DIFF value
0	0
1	-1, 1
2	-3, -2, 2, 3
3	-7..-4, 4..7
4	-15..-8, 8..15
5	-31..-16, 16..31
6	-63..-32, 32..63
7	-127..-64, 64..127
8	-255..-128, 128..255
9	-511..-256, 256..511
10	-1023..-512, 512..1023
11	-2047..-1024, 1024..2047

Table 2: Encoding of bit width codes

The table shows a set of bit widths, from 1 to 11, and the values that are represented in each bit width. So, for instance, bit width 1 represents -1 and 1 (as values 0b and 1b), and bit width 2 represents -3, -2, 2 and 3 (as values 00b, 01b, 10b and 11b), and so on. All the numbers from -2047 to 2047 are thus represented. The 11 bit codes are only used for the DC value, however, and 10 bits is enough to represent all the AC values.

If we encoded the amplitudes using the transform above, without any further information, it would be impossible to delimit the variable length codes and retrieve the data. We must precede these values with more information, in particular the bit width of the following code. Then we know how many bits to extract to find the amplitude. As the maximum code size is 11 bits, only four bits are needed to store the bit width of the following amplitude code. However, we can now look to run-length encode the zero values which may be present. So, in addition to the four bits for the bit-width of the amplitude code, we add another four bits to represent the number of zeros that preceded the amplitude value, up to a maximum of 15. So, for example, if we want to encode an amplitude value of -14, which has 3 zero values before it since the last non-zero amplitude, then we output a code 00110100b followed by 0001b. The upper nibble of the code (0011b) is for the 3 zeros, the lower nibble (0100b) indicates 4 bits of amplitude to follow, and then the 4 bits are 0001b since -14 is the second code of the 4 bit values, as per Table 2 above.

Since the maximum run length of zeros which we can represent in the code is 15, there is a special code "ZRL" with a value of 0xF0 to represent 16 zeros (15 zeros followed by 0 bits- which from table 2 is 0, making 16 zeros). These can be chained to represent as many zeros as required. There is also one other special values, 0x00, which represents "End of Block" (EOB). This terminates the codes for the array early, when all the remaining values are 0.

So we now have a set of run-length/bitwidth codes, followed by variable length codes for amplitude values. Are we ready to output these values yet? No! The run-length/bitwidth values themselves contain a lot of redundancy-for instance, early on in an array many of the amplitudes may not contain any zero values, and so all the codes will have 0000b as their run-length nibble, each requiring 4 bits to represent. So the run-length/bitwidth codes themselves need to be turned into more efficient variable length codes using a technique called Huffman Coding.

Huffman Coding

A detailed description of Huffman coding is not appropriate here. Suffice it to say that the method involves encoding the more likely values (i.e. ones repeated most often in a given set of data) with smaller codes, and the less likely (rarer values) with larger codes. More detail of Huffman coding, with an example is given in the Data Compression Article [3]. A real example of a Huffman tree for encoded values is shown in Figure 5 below:

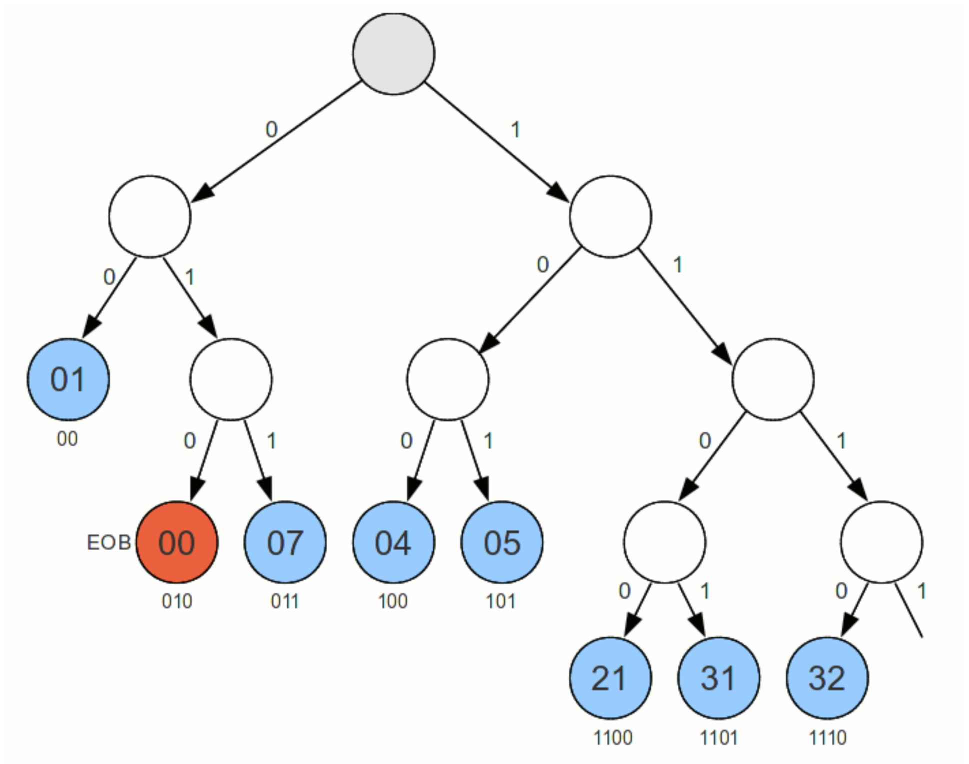

Figure 5: Huffman Tree Example

In the above figure is encoded a set of run-length/bitwidth values. The left branches of the tree are 0s, and the right branches 1. So, the first code we have (0x01) is encoded as a 2 bit code of 00b. This is the only 2 bit code (there are no 1 bit codes), and the next code (0x00, which happens to be EOB) following the tree path, is encoded as 010b. Similarly 3 other codes, before three other codes are represented as 4 bits. As Huffman codes are constructed such that no smaller code is a prefix of a larger code, these variable length codes can be output without further qualification, and uniquely decoded. As a consequence of this, there are some other interesting things to note here, which will come in handy for implementation.

For any given bit width set of Huffman codes, the codes (from left to right in the example figure) are an incrementing sequence. So, in Figure 5, the three bit codes are 2, 3, 4, and 5. Similarly for the 4 bits codes (12, 13 and 14). The very first code (whatever its width) is always 0. Another feature is that the first code of a given bit width is the previous bit width's endcode +1, shifted left 1. So the endcode of the 3 bit codes is 101b. Adding 1 (giving 110b), and shifting left 1, yields 1100b, which is the first code of the 4 bit codes. Not shown in the example, but if a bit width is skipped (e.g. if we went from 3 bit codes to 5 bit codes, without any 4 bit codes), then we shift 1 for each skipped bit width to give the first code of the next populated bit width part of the tree. These noted facts may seem esoteric now, but these will become very useful when implementing decoders in avoiding having to build full Huffman trees, and traversing them a bit at a time.

With this step we have finished encoding the data. There is a variable length Huffman code to represent the run-length/bitwidth information, followed by a variable length code representing the quantised DCT values. The Huffman codes themselves (the tree) is not discernible from the encoded data, and the tree must be encoded in the output, as a Huffman Table, just as the quantisation tables need to be. The quantisation tables organised by luminance and chrominance, and the Huffman tables are no different, but, in addition, there are tables for DC and AC values. The DC tables are usually much smaller than the AC tables, and require fewer (and smaller codes). The nature of the DC codes are different as well, with the upper nibble always being 0 (no run-length) and DC amplitude being a difference code. So encoding of the DC values separately makes sense.

Format

Having discussed the basic concepts, we can start looking in more detail at the actual organisation of data within a JPEG encoded image. There are many options specified in the JPEG standard [1], and we are not covering all of them here, but will look at enough of the format to be able to cope with the vast majority of data, and that match up against our needs to produce a decoder implementation. More details of all the format option can be found in Annex B of the standard.

In general, JPEG data is organised in a hierarchy, with markers delimiting sections of the format, which themselves may have marker delimited areas including data areas, tables, image info etc. A typical organisation is shown in Figure 6 below.

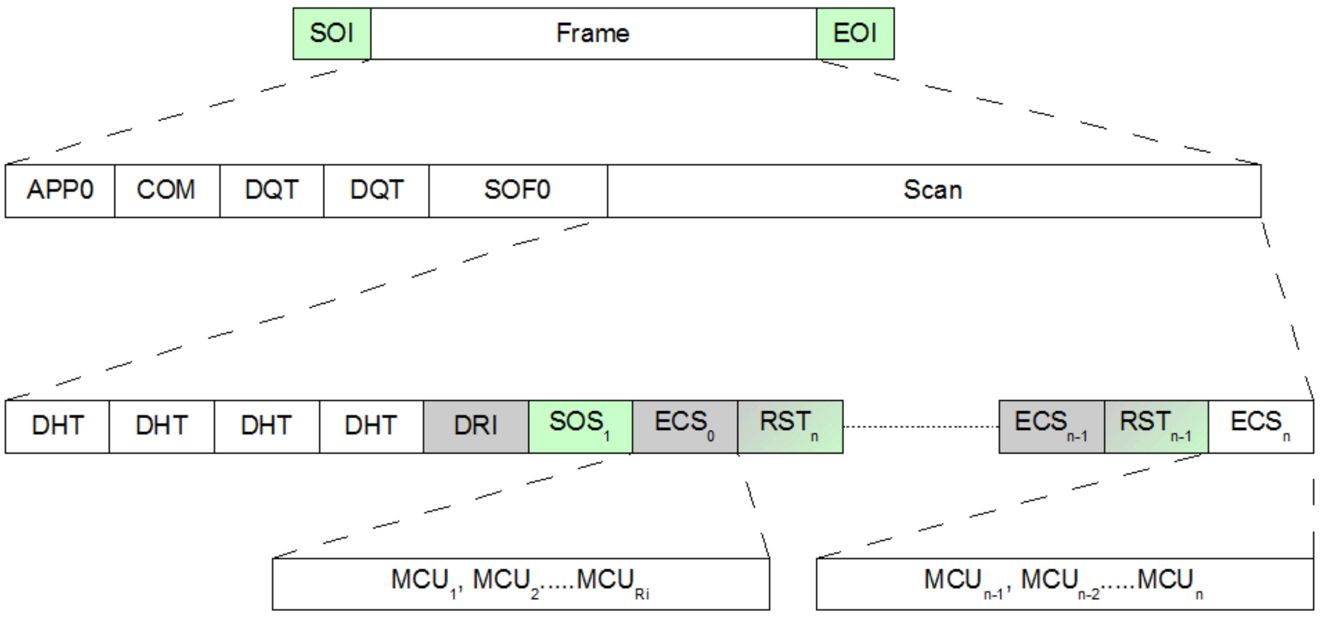

Figure 6: JPEG Format Organisation Example

The data is organised as a "Frame" between two markers: a start-of-image (SOI) and an end-of-image (EOI) marker. The frame itself is composed of various sections, with miscellaneous header and information, followed by a "start of frame" (SOF) header, and then the scan section itself. According to the JPEG standard [1] the sections before the SOF header are optional. As we shall see, the JFIF format insists on a least an application-0 (APP0) section. Shown in the example above is a comment (COM section), followed by two de-quantisation tables (DQT). It is unlikely that the quantisation tables are optional, despite the specification, and also there are other sections that can be added that are not shown above, mostly other application sections (1 through 15). These other application sections, as well as the comment sections can usually be ignored by a decoder, unless very specific information is known to be stored in them. The start-of-frame section is shown as an SOF0. This indicates a "baseline DCT" encoding of the JPEG data. SOF1 to SOF15 are for different encodings (such as progressive or lossless DCT). These formats are rare, and we will not be dealing with them here. Also, we have only shown a single scan in the figure, but mutiple scans can exist separated by a "define-number-of lines" (DNL) marker. Again, this is not relevant and not covered in this article.

The actual scan itself consists of a set of optional (according to [1]) miscellaneous data and tables, followed by "start of scan" (SOS) header, before reaching the data. The figure above shows four Huffman table sections (not really optional), and a "define-reset-interval (DRI-really optional), before the SOS header. A "define-reset-interval" (DRI) section, if it appears, defines how many "MCUs" (see paragraph below) are processed before a "reset marker" appears (RST). These markers define where the DC component running counts are reset (see above). The RSTn are used modulo 8, so that the first RST seen will be RST0, then the next RST1, and so on, until RST7, when the following RST will be RST0 once more. If no DRI/RST are inserted, then there is a single "entropy-coded-segment" (ECS) containing all the encoded image data.

The scan data itself is organised into "minimum-coded-units" (MCUs). These comprise of one or more 8x8 data sets. For greyscale images, this is simply a single encoded 8x8 luminance array. For colour images, this is luminance data followed by chrominance data (Cb then Cr), remembering that if sub-sampled (see above), there may be multiple Y arrays.

Now, both the JPEG [1] and JFIF [2] formats specify some ordering of the sections, and which may, or may not be present, *but* experience has taught that not all data encountered is completely compliant with these requirements. When constructing a decoder it is safest to assume that any of the sections may come in any order, so long as MCU data is at the end. This ensures that all the header information and tables are scanned before decoding the image data, but makes no other assumptions about the data. It is *very* likely that the frame header will preceded the scan header, but it is still more robust not to assume this in a decoder. If fussy, then make a switchable warning.

Another thing to note is that, although the example of Figure 6 shows multiple DQT and DHT segments, which is often the case, multiple quantisation and Huffman tables can be defined in a single segment. The only indication of this is that the length field of the segment is a multiple of what it would be if there were only a single table defined. See below for details of the table formats.

JFIF versus JPEG

The JFIF format does not substantially alter the basic allowed JPEG format, but does try and tie a few things down. For our purposes it does two things. Firstly it insists that the colour space by YCbCr (or just Y for greyscale). Pure JPEG allows other colour space encodings (which we're not supporting). Secondly it uses the application section APP0 to define some additional information, and insists that a JFIF APP0 section be the first data (though this is *not* strictly adhered to in my experience). This first APP0 section has an ID field of 5 bytes with the string "JFIF\0". The rest of the fields define a version, some density information and some definitions for and optional thumbnail. This thumbnail is housed in a second APP0 section, with an ID string of "JFXX\0"

We don't care too much about all this JFIF information, and can safely skip over these sections. The only interest might be to validate the data as JFIF (and thus guarantee the colour space encoding).

Markers

The markers in JPEG are used to delimit headers and sections or (in a few cases) stand alone as positional markers. For our limited discussion only a sub-set of all the markers are relevant, and an abbreviated table is given below to define these.

Symbol	Code	Description
SOF0	0xFFC0	Start-of-frame for baseline DCT
DHT	0xFFC4	Define Huffman Table
RSTn	0xFFD0 to 0xFFD7	Restart with modulo 8 count
SOI	0xFFD8	Start of image
EOI	0xFFD9	End of image
SOS	0xFFDA	Start of scan
DQT	0xFFDB	Define quantisation table
APP0	0xFFE0	Application segment 0
APP14	0xFFEE	Application segment 14
COM	0xFFFE	Comment

Table 3: Sub-set of JPEG Markers

Note that SOI, EOI and RSTn are all stand-alone markers, and do not indicate a following segment or header. As for the other markers not listed we might treat them all as invalid/unsupported (though this is somewhat conservative), or ignore them and skip over the segments and see if this works out. A mixture of the two is probably most sensible. So, skipping unrecognised application data is probably okay, but a non-zero start of frame indicates a non-baseline JPEG, and so is most likely a showstopper.

Headers

All the header segments follow the same basic pattern; a marker, followed by a 2 byte length and then the rest of the segment data as defined by the length. This length includes the two bytes of itself as well as the following data, but not the segment's marker. The only two header segments we're really interested, other than the table segments, are the start-of-frame and start-of-scan headers. These contain information to decode the data, such as image size, sub-sampling factors, component IDs and table mappings etc.

Parameter	bits	Values (baseline/supported)	Description
L	16	8 + 3xNf	Length
P	8	8	Sample precision
Y	16	0 to 65535	Number of lines
X	16	1 to 65535	Number of samples per line
Nf	8	1 or 3	Number of image components in frame
Ci	8	0 to 255	Component identifier
Hi	4	1 or 2	Horizontal sampling factor
Vi	4	1 or 2	Vertical sampling factor
Tqi	8	0 to 3	Quantisation destination selector

Table 4: SOF Header

Table 4 above shows the start of frame (SOF) segment format. For baseline DCT, the precision parameter (P) is always 8; i.e. 8 bit samples are used. The dimensions of the image are given in Y and X, whilst the number of components in the image is defined by Nf. This is either 1 for a greyscale image, or 3 for a colour image. Depending on the Nf value, the following parameters may be repeated to give 3 sets-one for each component. The Ci parameter is an identifier for the component (and is usually 1, 2 or 3 for colour images, but need not be), whilst the Hi and Vi fields give the sub-sampling factors. These should be either 1 or 2, otherwise a non-supported sub-sampling is defined. For multiple components (colour), the values of Hi and Vi should be identical for all components to make any sense. Finally the Tq field identifies which quantisation table is to be used for the component. Four quantisation spaces are available, and will have an identifier associated with them. This field indicates that the table to use should be that matching Tq. Note that the tables are likely to have IDs of 0, 1, 2 and 3, but they do not have to be.

Parameter	bits	Values (baseline/supported)	Description
L	16	6 + 2xNs	Length
Ns	8	1 or 3	Number of image components in scan
Csj	8	0 to 255	Scan Component Selector
Tdj	4	0 or 1	DC Huffman table selector
Taj	4	0 or 1	AC Huffman table selector
Ss	8	0	Always 0-ignore
Se	8	63	Always 63-ignore
Ah	4	0	Always 0-ignore
Al	4	0	Always 0-ignore

Table 5: SOS Header

Table 5 shows the start of scan (SOS) segment format. The Ns field defines the number of image components in scan. This should be the same as Nf in the SOF header. The following three fields (in blue) are then repeated Ns times, one for each component. Csj defines which of the frame components this one shall be. This *should* be the same order as defined in the SOF, but need not be. So, usually, we have SOF component IDs of 1, 2 and 3, in that order, with the same 1, 2 and 3 order in the Cs IDs. But it could be that some encoder decides to mix them up, and define 2, 3 and 1-and so the first data is ID 2, and uses the SOF information defined second, and so on. (I have never seen this!) The Td parameter selects the Huffman table to use for the DC values, whilst the Ta selects the table for the AC values, for this component, in much the same way as for quantisation table selection in the frame header.

The (non-repeated) remaining parameters are not relevant to baseline JPEG data, and so are not discussed here. They can simply be skipped over.

Quantisation Table

The quantisation tables are simply the 64 divisors used to scale the DCT amplitudes (see above), and the DQT segment simply stores these numbers with a little extra information. Table 6 below shows the format of a DQT segment.

Parameter	bits	Values (baseline/supported)	Description
L	16	2 + 65 x num_of_tables	Length
Pq	4	0	Quantisation table element precision. Always 0 (for 8 bit)
Tq	4	0 to 3	Quantisation table destination identifier.
Qk	8	1 to 255	Quantisation table element

Table 6: Quantisation Table Format

After the length bytes follows 1 or more tables, consisting of 65 bytes. Multiple tables within one DQT segment are only indicated by the size of the length parameter. Tables can, however, be stored in separate segments. The first byte of the table consists of a couple of parameters, the first of which, Pq, is always 0 for baseline JPEG data. The Tq parameter gives the table destination ID, selecting 1 of 4 table buffers. The following 64 values are the quantisation values, serialised with the zig-zag pattern (see figure 4 in the quantisation section above). This 65 byte sequence is repeated for other tables, if present in the same segment.

Huffman Table

The Huffman table segments are similar to quantisation table segments, in that they are a length followed by 1 or more table data portions, with the length being the only indicator of multiple table content. As the Huffman tables are not of a fixed size though, this is only discernible after parsing each table, and checking the remaining count. Table 7 below shows the table format.

Parameter	bits	Values (baseline/supported)	Description
L	16	2 + ((17 + mi) x num_of_tables)	Length
Tc	4	0 or 1	Table class (0 for DC, 1 for AC)
Th	4	0 or 1	Table destination ID (2 tables per class)
Lx	8	0 to 255	Length of codes for bit width x (16 values for x = 1 to 16)
Vij	8	0 to 255	The jth code associated with bit width i

Table 7: Huffman Table Format

Each table within the segment starts with the Tc and Th parameters. The first specifies whether a DC table or an AC table. The second gives (for baseline DCT) one of two table destinations. This give four possible tables-two for DC and two for AC. The DC tables will almost certainly be much smaller than the AC tables, and so the storage needed for the two classes can be different. The Tc/Th parameter byte is always followed by 16 length values. These specify the number of codes for each bit width in the table, start from bit 1, up to bit width 16, and these can be 0. This is then followed by the codes to be stored in the table, Vij. The number of codes is determined by the length values, and will equal the total values of all 16 length added together. Figure 7 below shows a single table layout example which matches the Huffman tree of figure 5 in the Huffman Coding section above.

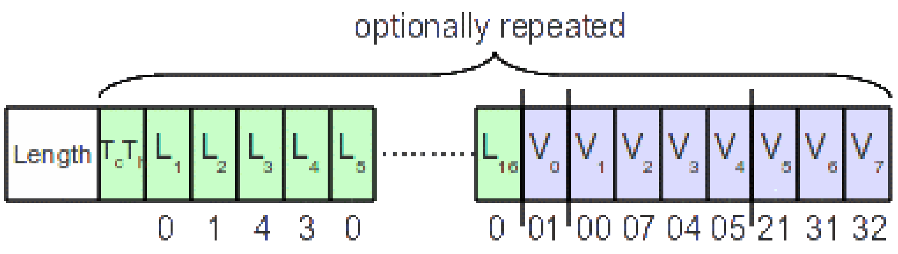

Figure 7: Huffman Table Length and Value Example

In the Huffman tree of Figure 5, the first code was a 2 bit code, storing a code of 01h. Therefore, in the above example table, L1 is set to 0, as there are no 1 bit codes. L2 is set to 1, as the 01h code is the only code of bit width 2. Four bit width 3 codes are then defined, so L3 is set to 4, and the codes for 3 bits (00h, 07h 04h and 05h) are placed as V1 through V4. Finally, three 4 bit codes are used, L4 is set to 3, and the stored codes (21h, 31h and 32h) a placed in V5 through V7. Thus the 8 codes are placed in the DHT segment, and the length values delimiting them add up to 8 also, as expected.

Other Segments

Of the other segments (e.g. APP, COM, DHP etc.) only one is of interest in decoding JPEG data. This is the "define-restart-interval" (DRI). Table 8 shows its format.

Parameter	bits	Values (baseline/supported)	Description
L	16	4	Length
Ri	16	0 to 65535	Restart Interval

Table 8: DRI Segment Format

This segment has only one parameter, Ri, which defines the number of MCUs (minimum coding units—see below) between restart intervals. What this means is that for every Ri MCUs processed, a RSTn marker is expected. This basically tells the decoder to reset its DC differential values back to 0 (see end of DCT section above).

Actually, one might notice that there is unnecessary redundancy here. Having defined a DRI, one does not need the RSTn markers. Or, alternatively, if the RSTn markers are present, then there is no need to define an interval. Anyway, the format states that this is how things are, and so it's no use arguing now.

Scan data

The scan data itself is simply the encoded data stream of the 8x8 arrays, transformed and encoded, as described in the earlier concept sections above. They are not delimited, unless a restart interval was specified, in which case RSTn markers are placed between MCUs.

The data is organised in to MCUs (minimum coding units). This is just a bundle of arrays associated with each other. So, for pure grey scale, an MCU is a single Y luminance encoded array. For colour images, without sub-sampling, this is a triplet of luminance followed by the 2 chrominance encoded arrays, in the order "Y Cb Cr". When sub-sampled, all the Y encoded arrays are output first, followed by the averaged chrominance data. The order of the luminance data is from left to right and then (if vertically sub-sampled) upper to lower. For 4:2:0 subsampled data this gives an ordering of "Y(x,y), Y(x+1,y), Y(x,y+1), Y(x+1,y+1) Cb, Cr".

Conclusions

Described above are the main concepts involved in encoding and decoding JPEG data, and some details for the format. To summarise these steps:

- Sub-divide image into 8x8 sections
- If a colour image, separate into luminance and chrominance channels (YCbCr)
- Optional horizontal and vertical chrominance sub-sampling
- Discrete Cosine Transform (DCT)
- Turn DC values into difference values
- Quantise amplitudes
- Variable length encode non-zero amplitudes
- Prefix with zeros run-length and amplitude bit width
- Huffman encode run-length/bitwidth codes

Decoding is simply the reverse of encoding. The format section detailed how the data is represented in JPEG, with the storing of the quantisation tables and Huffman tables before the actual image scan data.

Not all concepts of JPEG were covered. In particular major concepts like arithmetic coding (an alternative to Huffman coding) or progressive JPEG were avoided. The concepts here are limited to those relevant to the implementation articles to follow, but it's hoped that enough ground has been covered to allow the reader to tackle JPEG data with some confidence, and have a handle to research the missing concepts for themselves. Most data that the author has encountered would be covered by the above discussion. The most common exception is progressive JPEG files.

References

- [1] JPEG Standard (JPEG ISO/IEC 10918-1 ITU-T Recommendation T.81)
- [2] JPEG File Interchange Format Specification v1.02
- [3] Notes on Data Compression and Related Topics: Huffman Encoding.