# Notes on Data Compression

Simon Southwell

February 2022

# Preface

This document brings together five articles written in January and February 2022, and published on LinkedIn, that cover data compression methods, from lossless algorithm review (part 1), a more in-depth look at the LZW algorithm (part 2), a software implementation (part 3), the JPEG lossy image algorithm and format (part 4), and a JFIF/JPEG software decoder implementation (part 5).

Simon Southwell
Cambridge, UK
February 2022

# Contents

# Part 1: Overview

## Introduction

Recently, the subject of data compression unexpectedly has come up from several people I have contact with. It is something I was involved with quite some time ago, making ASIC designs that included lossless data compression logic for mass storage devices. Since this has all been brought back to mind, I went over some material I wrote a little while ago and thought it might be useful to revisit this material which may be of some use. The original material is somewhat long for a single article so the document is divided into parts and this is the first part on data compression which will review the various popular lossless methods in outline. In this first part we will look at run-length encoding and two algorithms that use 'models' of symbol probabilities; Huffman and Arithmetic. The next few parts of the document will cover, in a little more detail, the LZ class of algorithms and look at the 'dictionary' based LZW algorithm as a case study, with an implementation in C examined. Then the lossy image compression algorithm of JPEG and the JFIF format will be covered, again, with an implementation of a decoder in C++ to illustrate the main points.

As alluded to above, there are two major classes of data compression techniques:

- Lossy
- Lossless

Lossy algorithms act on data which has redundancy in information which is 'not critical' in order for the data to retain 'meaning'. This usually means data that is to be interpreted by human perception. The most classic cases of this type of data are images (either still or moving pictures) and audio (either voice or music). Of the former type, an example is the JPEG (Joint Photographic Expert Group) standard. This takes advantage of the fact that high frequency components in an image are much less important (for human interpretation) than low frequency components, and it thus reduces (or loses) some of the information in this part of the data content in order to compress the image into a smaller space. Lossy algorithms are also characterised by achieving high data compression figures, as one might expect if data can be thrown away. But this all depends on just how much data can tolerably be lost. After all, in the limit, continued compression will result in just a single bit, but this carries no interpretable data whatsoever.

By cutting down this data to smaller sizes, reduced storage requirements are not the only benefit. A fair-sized image may be 2MBytes when raw data but might compress

down to 50KBytes. Imagine downloading this image over the web at, say, 5KBytes/S (on a good day). The original image would take nearly three and a half minutes. Compressed, however, and the image is transferred in twenty seconds. Lossless algorithms, by contrast, do not 'lose' one single bit of data, and any compressed file is decodable to its original form. Here the algorithms work on data redundancy due to 'inefficient' representation—effectively, on average, more than 1 bit is used to store 1 bit of information . If the data is 100% efficient, then no lossless data compression technique can compress this any further and will in fact expand the data slightly due to its own coding overheads. However, much data does have this type of redundancy in it; databases with largely empty search table, source code with repeated spaces and frequent key word strings, English text and so on.

In the world of information technology, compression of data is desirable for two reasons. Firstly, more data can be stored per unit of media (reducing the cost of that media) and secondly the transfer rate to the device is increased in proportion to the compression rate (with a sufficiently fast compression device, and a large enough buffer), thus reducing the storage/transmission time for a given unit of data. So, which algorithm to choose? Which is the best? There are certainly a few to choose from and, as with many engineering problems, there are trade-offs to be made before a final implementation is decided upon. These problems can be as diverse as the maximum amount of dedicated memory that's required or who owns a particular patent and dictates the licencing agreements for that patent. In the following sections we will look at some lossless algorithms in a general manner, followed (in the later parts) by a more detailed look at how two of these algorithms may be turned into algorithms that can (and are) used in embedded hardware compression 'engines'.

# Lossless Data Compression Techniques

Lossless compression algorithms (sometimes referred to as noiseless or reversible algorithms) are characterised by the ability to retrieve the pre-encoded data with 100% accuracy. They all attempt to re-encode data in order to remove redundancy, and this implies that data with no redundancy cannot be compressed by these (or any other) techniques without some loss of information. Indeed all these algorithms will have some degree of overhead in their encoding methods and will therefore actually expand non-redundant data. An example of such data might be a data set that has already been compressed.

Lossless data compression techniques do not always remove redundancy with 100% efficiency, and often algorithms with different emphasis are used together to give

more efficient compression. Or perhaps some pre-processing of data (which doesn't compress) is used in order to make the compression more favourable for the compression technique. E.g. the Barrow-Wheeler transform (BWT).

Many algorithms exist with broadly similar characteristics, and sub-variants of these abound in countless measure, each trying to squeeze out the last drop of efficiency. In the following discussions, various different types of algorithm are given a perusal in order to make comparisons, and to give a context to two particular schemes, which we will look at most fully. As had been stated before, one may implement data compression as software running on a CPU, and much has been written on software implementation (see Nelson and Gailly "The Data Compression Book", 1996 for an excellent account). To give a different flavour, I want to look at these techniques with a hardware implementation bent. Hopefully this will give a different perspective on the subject, but it will also place different constraints on us that might not be applicable to a software implementation. For example, we are likely to want to compress data in excess of 40Mbytes/s, but the amount of memory available to might be less than 1Kbyte. So what gives! Well, we will have to design an ASIC or FPGA, no small task, and we are likely to fall short in compression ratio compared with a good software compression program; but a Formula 1 racing car might not be comfortable, but it sure leaves a fat Mercedes standing. So let's actually look at our first strategy-run-length encoding.

## Run Length Encoding

This is probably the simplest algorithm to understand, though it is quite powerful for particular types of data that have certain characteristics, as we shall see. This algorithm works by substituting a run of repeated characters with a single instance of the repeated character followed by a count indicating the repetition number. So, for example, suppose we had a data set consisting of the following sequence of characters from a 3-character alphabet:

**AAAABBBBBBBCCCCCCCAAACCCCCCAAAAAAA**

This might be encoded as:

**A4B7C7A3C6A7**

The new encoding has added some extra 'symbols' for encoding the run length, but in this very simple scheme we have compressed the character sequence by 34/12 = 2.8. Of course, the scheme shown is too simple to be practical (you could make it more efficient, for instance, by avoiding having to encode single characters as '1').

The compression ratio also assumes that we didn't need to add any more 'bits' in order to insert run length characters, which is the same as saying that the original alphabet was grossly inefficiently encoded. A 3-character alphabet only need 2 bits to encode each character. Assuming that we allow run lengths of 1 to 10, then the run lengths plus the original alphabet can all be encoded in 4-bit characters. So, the original message has 34 × 2 = 68 bits. The compressed message has 12 × 4 = 48 bits, a compression ratio of approximately 1.42. So you see, practicalities of encoding limit simplistic schemes, and one must be careful to understand what one is compressing. It is very easy to compress data with high redundancy, but data is rarely that forthcoming.

Even in a practical scheme, you should be able to see that it will only be good if the data is compresses has a high instance of repetition. This characteristic is not shared universally between data sets and run length encoding is often restricted to data which are essentially bit streams (hence have a two-character alphabet, 1 and 0) and which also has a high likelihood of repetition, such as one might find in graphical data; e.g. bitmaps. It is also sometimes used as a post processing algorithm of other compression techniques where their output has some of the favourable characteristics of run length encoding.

## Huffman Encoding

This scheme ("*A Method for the Construction of Minimum-Redundancy Codes*", Huffman, 1952) is a 'variable-length' encoding. It is still widely used today but can be quite complicated to implement. The general aim is to have a set of uniquely identifiable 'codewords,' which have different lengths, and then assign the smallest codewords to represent the most frequent characters in the data set. Of course, identifying which characters (or more generally, symbols) occur most frequently is not an easy task. We will need a 'model' of this character frequency; i.e. a table of probabilities for the occurrence of any given character. There are many ways to do this, but the simplest to understand is where the entire data set is processed, counting character occurrences as we go. The table is then constructed as 'number of characters X divided by total number of characters in data set.' Actually, for the method we're about to describe, the character totals are all we need, as only their relative rank is important and we want to avoid any non-integer arithmetic in an implementation. So, we can easily rank each character in terms of its frequency for a given data set. Now we need a coding scheme with the following attributes. It is made up of a variable length set of codes, and each code is uniquely identifiable from all the others.

It isn't necessarily obvious how we might do this. One can't simply assign low numbered codes with less bits, as in `0`, `1`, `10`, `11`, `100` etc. Given a bit stream of, say, `010101001101010` the coding scheme could yield many different interpretations. The way to proceed is to have a coding scheme whereby no code is a prefix of another code. For example: `00`, `10`, `11`, `010`, `011` etc. Here, two-bit codes are `00`, `10` or `11`. This leaves `01` as an unused two bit number. All other codes must use this sequence as the first two bits of their code. By expanding a scheme like this for a more practical number of codewords (such as 256) an implementation can be realised. Note, however, that it is not possible to have a code of a maximum N bits encode 2N codewords, and the longer codewords will 'expand' the characters they encode. Some characters must be more frequently encountered than others.

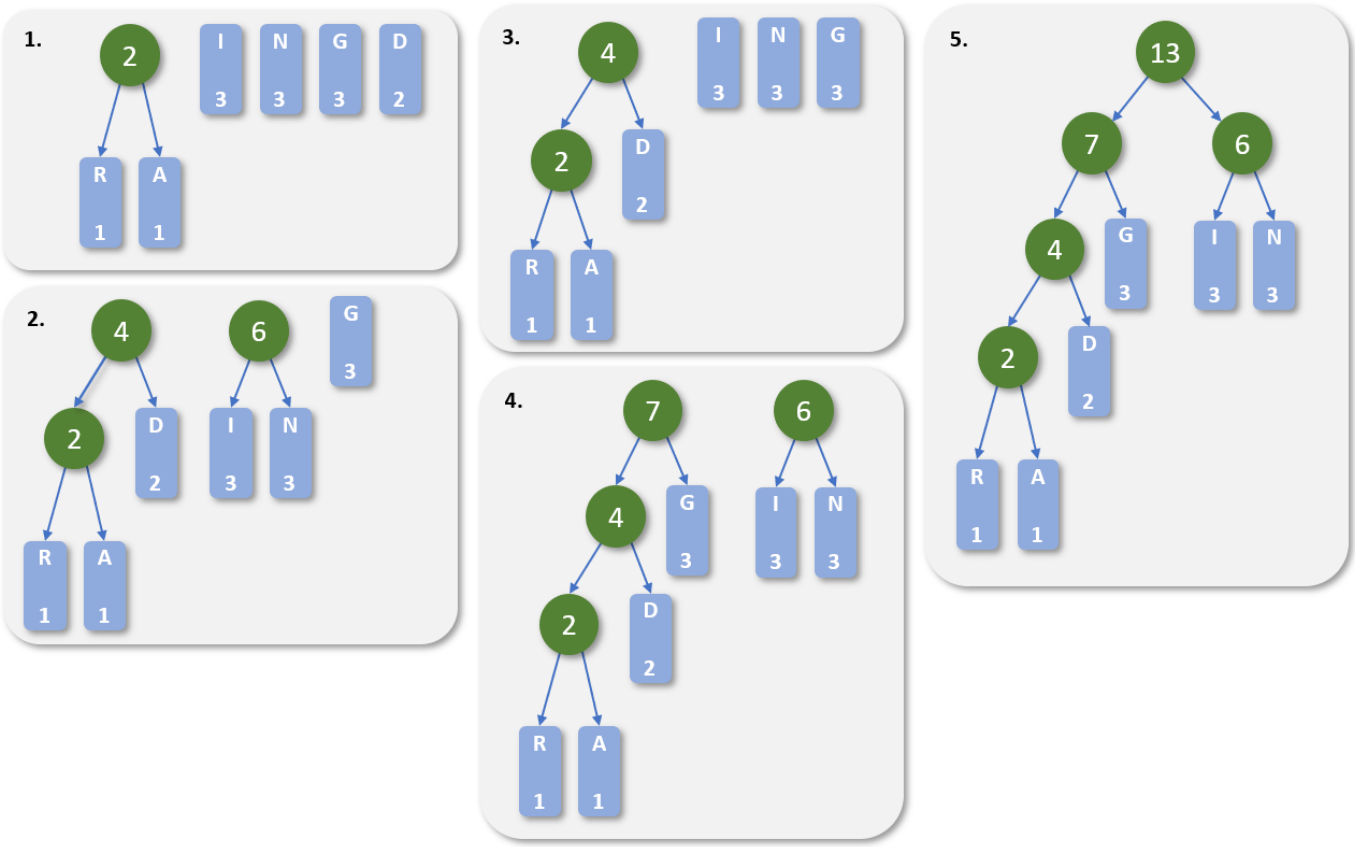
An algorithm can be described which automatically sort the codes for each character, assigning the smallest codes to the most frequent characters. The algorithm is going to build a Huffman Tree which can then be read off to find the code to use. First the frequency counts for each character used in the data set is calculated, as described above. The two lowest ranked (i.e. lowest counts) characters are 'paired,' forming a 'node' of a tree, where the new node has a count which is the sum of the two characters, and it has 'branches' to each of the two paired characters. These characters will no longer be considered in subsequent pairings. If more than one choice is available when pairing, then it is totally arbitrary which choice is made. The next two lowest values are paired in the same way, but any new nodes formed, with their summed counts, must be considered as if they were characters. This procedure continues until only one node exists at the top of a tree which should have a count total equal to the number of characters in the data set being considered. The 'leaves' of the tree should be the characters that appear in the data set.

Codes can now be assigned as follows: starting from the top of the tree, to get to a character one must move down a branch, left or right, to reach a given character. We can assign (arbitrarily) 0 to mean left and 1 to mean right (though the reverse works just as well). So the path from the top of the tree to the leaf containing the character maps out a sequence of 0s and 1s for the left or right transversals. These are the bits which make up the code for that character. All the other characters are assigned codes in the same manner.

Let's have an example. Suppose that we wish to encode the short message `RINGADINGDING`. The frequency table for this example is as follows:

| **R** | **I** | **N** | **G** | **A** | **D** |
|---|---|---|---|---|---|
| 1 | 3 | 3 | 3 | 1 | 2 |

The sequence of building a Huffman tree with the lowest frequency pairings is shown in the figure below.




The final tree now gives us the coding for the characters. Choosing a left branch as 0, and a right branch as 1, we have the codes of R=0000, A=0001, D=001, G=01, I=10 and N=11. As you should be able to see, the most frequent characters have been assigned the shortest codes, and the shorter codes are not prefixes of the longer ones. The message (`RINGADINGDING`) is thus encoded as:

```
00001011010001001101101001101101
```

This is 32 bits which gives a compression ratio (assuming 8-bit ASCII characters in the original message) 48/32 = 1.5:1 (not bad). However, if we received the above encoded message how would we know what the codes were and how they had been assigned. Unfortunately, in our simple scheme, we would also have to pass over the code table in order for the message to be decoded. In our example, this would be an enormous overhead. As data sets get bigger the overhead diminishes and it becomes more practical a method.

It might be possible to agree a fixed coding table, so that both sender and receiver knew the code mappings and a table needn't be sent. This is fine if the same message is always sent, but every new message will have a different frequency table,

and thus a different Huffman tree with its code assignments. Data of similar origin might have similar frequency distributions, for example English text. So we might have an English text code mapping and could use this if all messages are in English. The compression rate will only be as good as the table, and so care would need to be taken when constructing this 'model.' Many examples of English would need to be used, and unless the message has exactly the frequency distribution of the fixed table we are using, the compression ratio will always fall short of the ideal. This is why you are unlikely to come across this kind of pre-calculated table in any implementation. Constructing and sending the codes for the particular method is more likely, and the penalty in speed efficiency of reading the data twice (once for the mapping, once for the encoding) is borne.

An alternative is to start with all characters being equal probability. As new characters are encoded, the frequency distribution is updated 'on the fly.' As the character ranks change, so do the code assignments. The receiver has the same algorithm available to it. As it decodes data it too keeps relative ranking tracked and adjusts the code mappings in exactly the same manner as the encoder. This has the advantage of obviating the need to transmit the coding table, and the encoded data is not read twice in order to construct the codes. This will still not be as efficient (compression ratio wise) as the Huffman tree construction mentioned above, but is an improvement on fixed tables, with the speed advantages mention, and it is used widely with Huffman and other variable-length coding schemes.

## Arithmetic Coding

Huffman coding, mentioned in the previous section, has been described as 'best,' assuming the two-pass encoding described above. This is true only if we use codes that are an integer number of bits and are fixed. A related method is Arithmetic coding ("*Arithmetic coding for data compression*" Witten et. al. 1987). With this method codes can be stored in fractions of a bit. The method will achieve at least as good a result as Huffman coding but will often beat it. The penalty for this gain, as with so many things in engineering, is increased complexity.

The basis for the algorithm is to find a unique fractional number for the message being encoded, based on the probability of the symbols appearing in the exact order as in the message. It won't be an exact probability measure, as two messages may have exactly the same probability. As with Huffman encoding, each symbol has a probability of appearing in the message, which is the frequency of the symbol divided by the length of the message. So, for example, if the symbol '#' appears 5 times in a 20-symbol message, its probability is 5/20 = 0.25. For a message to be

encoded, all its symbols have their probability calculated in such a manner, with the totals, hopefully, adding to 1 (i.e. there is a 100% chance of finding a symbol that is somewhere in the message). Next, the probability space is divided up into ranges matching the symbol probabilities, stacked on top of each other, as it were. E.g. If a message has a 3-symbol alphabet (A1, A2 and A3, say), and the probabilities for the symbols are 0.25, 0.45 and 0.30, then we would divide the range of 0 to 1 into 0 to 0.25, 0.25 to 0.7 and 0.7 to 1.0. When the first symbol arrives for encoding, we select the range correspondingly (so if it were A2, then the range is 0.25 to 0.7). This new range is now our working range and is divided into the same proportional segments as was the 0 to 1.0 range initially. When the next symbol is encoded, we modify the current range to be the subrange of the first symbols range. So, if the next symbol is A1, then we need the range 0 to 0.25 of the range 0.25 to 0.7, which is 0.25 to 0.3625. This continues for the entire message. The range get narrower, and the decimal fractions get longer. When the entire message is encoded, we can pick one of the range bounds as the encoded message (usually the lower bound).

Let's have a more extended example to illustrate these ideas. We wish to encode the message:

**RINTINTIN.**

(10 characters, including the full-stop.) Firstly we work out the symbol probabilities (which I have made easier by choosing a 10-symbol message).

| Symbol | Probability | Lower Bound | Upper Bound |
|---|---|---|---|
| R | 0.1 | 0.9 | 1.0 |
| I | 0.3 | 0.6 | 0.9 |
| N | 0.3 | 0.3 | 0.6 |
| T | 0.2 | 0.1 | 0.3 |
| . | 0.1 | 0 | 0.1 |

so shown in the table are the range allocations for the symbols, with the upper and lower bound difference equalling the symbols' probabilities. It should be noted that it is arbitrary the order in which the symbols probabilities are allocated, though, of course, the encoder and decoder must agree on this point.

So, to start encoding the message, we take the first symbol 'R', and modify the current range to be 0.9 to 1.0. The final encoding will have a value which must lie between (or possibly on) these two values. The next symbol is 'I', with a range of 0.6 to 0.9. We then calculate what this range is within the previously calculated range. So, 0.6 of the way between 0.9 and 1.0 is (0.9 + ((1.0 - 0.9) × 0.6) = 0.96. Similarly, 0.9 of

the way between 0.9 and 1.0 is 0.99. So we now know the message values lies between 0.96 and 0.99. We have narrowed the range of possible numbers for our message. This procedure is carried through for the rest of the message's symbols, with the range becoming more and more restricted. The table below shows the results for the entire example message.

| Symbol | Lower Bound | Upper Bound | Difference |
|---|---|---|---|
| R | 0.9 | 1.0 | 0.1 |
| I | 0.96 | 0.99 | 0.03 |
| N | 0.969 | 0.978 | 0.009 |
| T | 0.9699 | 0.9717 | 0.0018 |
| I | 0.97098 | 0.97152 | 0.00054 |
| N | 0.971142 | 0.971304 | 0.000162 |
| T | 0.9711582 | 0.9711906 | 0.0000324 |
| I | 0.97117764 | 0.97118736 | 0.00000972 |
| N | 0.971180556 | 0.971183472 | 0.000002916 |
| . | 0.9711805560 | 0.9711808476 | 0.0000002916 |

As can be seen, the resulting numbers converge toward a particular point, with the difference between the upper and lower bounds becoming smaller and smaller. The final lower bound (0.9711805560) can now be used to uniquely identify the message. To decode the message, the reverse is applied. Since 0.9711805560 lies between 0.9 and 1.0 we know that the first symbol must be 'R'. If we now remove the lower bound values from this number (i.e. 0.9) and divide by the probability for 'R' (0.1), (that is, ((0.9711805560 - 0.9)/0.1), then we get the value 0.71181. Since this lies between 0.6 and 0.9, then the next symbol must be 'I'. This process is repeated to decode the entire message.

Now you are probably thinking that not only is this method more complicated than Huffman encoding, but it is very much more complicated than Huffman coding, and is the effort worth it? We have used multiplication and division, with large decimal fractions with many significant figures. Can we even handle this in a computer program or an ASIC? We can, fortunately, and using only integer arithmetic and minimal storage of bits. Firstly, any decimal fraction can be represented as a binary fraction, as one might expect, and, like floating point numbers, decimal places may be inferred, effectively scaling the numbers to be integers. Also, from the table may be noticed that, once a significant figure is the same in both the upper and lower bound values, then it will remain at this value for the rest of the encoding. This means that, when this occurs, we can output that value (or bit), rescale the remaining bits, and continue. With all this it is possible to get sufficient resolution with just 16-

bit arithmetic . There is one problem where the value may be converging to a point where (equivalently in decimal) the lower bound values are ...9999... and the upper bound values are ...0000... . This can be solved by 'borrowing' in a similar fashion in arithmetic subtraction. It is mathematically provable (though not by me) that 1.0 = 0.999999... recurring. We can substitute the zeros with this, so long as we remember, and correct the situation once the convergence has resolved one way or the other (i.e. on the ...9999... side or the ...0000... side).

So, we can solve some of the practical difficulties, and make this a viable algorithm for use in software or hardware systems. Having said this, the implementation is still not as simple as other alternatives, and this may be why Arithmetic Coding hasn't caught on as one might expect. That isn't to say it hasn't been used. IBM introduced and Arithmetic codec (coder/decoder) implementing a variant called IRDC (Increased Data Recording Capacity—it was used, originally in a tape storage device).

## Models

The Huffman and Arithmetic Coding techniques outlined above both require a table of symbol probabilities—a 'Model.' This can be a 100% accurate model, as used in the examples for these methods, but which requires a time consuming first pass over the data in order to construct the mode, which also needs to be transmitted (or stored) with the data in order to reconstruct the message. Adaptive methods were alluded to in the Huffman discussion. These have the advantage of needing no first pass of the data, and the model may be reconstructed on decoding, obviating the need for transmission or storage of that model with the message. The price paid is decreased compression ratios.

Models, ideally, should accurately reflect the probability distribution of the symbol occurrences, which, if compression is to occur should be non-uniform. In the above discussions, it has been assumed that a symbol's probability is fixed. This was calculated based on the occurrences of the symbols divided by the message length. This takes no account of any other symbol in the message. This is a zero-order model. A symbol's probability does, in fact, jump around quite a bit. For example, a text file may have a space character after every word, which might be a probability of 1 in 10, or 0.1. If the current character is a full-stop, then the probability for a space might rise to 0.7, say. By looking at the context of the character, i.e. the symbols occurring around it, a better statistical model may be obtained. The order of the model indicates the number of symbols used in determining the current symbol's context. So a 2nd- order model will use the two previous symbols in order to calculate the current symbols probability. However, as the model order goes up, so

does the memory requirements in order to store statistics for each context. In a zero-order model, with 8-bit symbols, only 256 statistics need to be kept. A 1st order model needs 65536 and a 2nd order model needs 16Mwords. I.e. it quickly gets very silly, and managing the resource also becomes cumbersome, having to search through larger amounts of data, and updating (in adaptive methods) greater table sizes. There is another way; the LZ adaptive coding class of algorithms which we shall look at in the next part.

# Part 2: The LZW Algorithm

## Introduction

In the previous part I did a quick review of some lossless data compression techniques, including run-length, Huffman, and Arithmetic coding. We will continue this look, in this second part, at lossless techniques with a discussion of the Lempel-Ziv algorithms before ramping things up a little and focussing in some detail on the LZW variant. An implementation of the LZW algorithm will be covered in the part that follows this one.

### LZ Adaptive Coding

The Lempel-Ziv family of compression techniques are what are known as 'substitutional' compression algorithms. They are characterised by the use of 'codewords' to substitute for 'strings' of symbols, which then reference a store to reconstruct the strings on decoding. The methods have some advantages over other methods in that they are 'universal'; i.e. they make no assumption about the data sets symbol probability distributions and are inherently adaptive. They also automatically encode the 'model' table within the encoded data, so it can be reconstructed by the decoder, without additional data transmission.

There are two main flavours of LZ algorithm; LZ1 (also known as LZ77—"*A Universal Algorithm for Sequential Data Compression*", Ziv et. al., 1977) and LZ2 (also known as LZ78—"*Compression of Individual Sequences via Variable-Rate Coding*", Ziv et. al., 1978). The first of these is conceptually the simplest to understand but can be slightly more complicated to implement in software (or indeed logic). In brief, the last $n$ symbols are kept in a buffer. As new symbols arrive, they are appended to the current 'string' of symbols. If the string is in the buffer, then the string is maintained and the buffer updated with the new symbol. Once the string has grown to a point where it is no longer in the buffer, the pre-appended string (that was in the buffer) is substituted for a code which consists of a pointer into the buffer where the string occurs, and a run-length indicating how long the string was. The symbol that caused the search to fail now becomes the single symbol string, it is appended to the buffer, and the process continues. Since only the last $n$ symbols are remembered, the buffer can be thought of as a window on the data, $n$ symbols wide, sliding along the data set. Hence you will often find this method called the Sliding Window technique. The buffer is a history of the last $n$ symbols, and so is often referred to as a History Buffer. In its purest form all strings are encoded as pointer/run-length codes, but this is

inefficient if the string is a single symbol. Variants might allow single characters to be encoded as 'literals', but then the pointer/run-length codes must be distinguishable from literals. So this is quite a simple method. The difficulty is in determining whether a particular string is in the history buffer, and ensuring the search is exhaustive. This method is, in my experience, easier to do well in logic than in software, where Context Addressable Memory can be implemented, if not easily, at least with some confidence. More on this later.

The second method (LZ2) is similar to LZ1 in its substitutional nature but uses a 'dictionary' of strings which have previously been seen in the input data, with codewords making references into the dictionary's list of strings, or 'entries.' These codewords are then substituted in place of the strings of symbols.

Initially the dictionary contains only the single symbol strings (e.g. values 0 to 255 for 8-bit bytes). As symbols arrive, they are appended to the current string, and the dictionary is searched to see whether it has an entry for that string. If it does, then a new symbol is appended to the current string. This continues until no entry is found in the dictionary for the current string. When this happens a new entry is assigned a codeword, and the string placed in the dictionary entry for that codeword. The codeword for the last matching string is output, and in the purest LZ2 form, the symbol causing the mismatch is also output. However, in the most popular variant, due to Welch ("A Technique for High Performance Data Compress", Welch, 1984), called LZW, the leftover symbol becomes the current single symbol string, and the process continues.

So this rounds up some major algorithms (excluding run-length encoding, rarely used on its own), implemented in varying degrees in both software and logic implementations. Next, I want to look in more detail at one of the more popular LZ algorithms and look at LZW, as a case study, and at what it takes to make it a fully specified and operational algorithm which can be implemented as a logic codec or a software implementation.

# The Lempel-Ziv-Welch (LZW) Compression Algorithm

As has been mentioned in the section above, the LZ2 algorithm, in particular the LZW variant, has a dictionary with a number of entries. Each entry has a codeword associated with it, with a unique value. Each entry refers to a string of symbols that have either been seen in the incoming data stream before or are the single symbol strings that the dictionary was initialised with.

From now on, symbols will be 8-bit bytes, with values 0 to 255. It doesn't matter if these bytes represent ASCII codes, or object data or whatever. The entries, with their associated codewords, which represent the single byte strings are known as root entries (or root codewords). To make this algorithm practical we will have to start imposing some restrictions on what is allowed. For example, we cannot keep building new entries in our dictionary indefinitely and it is usual, though not universal, to restrict codewords to be 12 bits wide, giving a possible 4096 codewords, 256 of which must be the root codewords. The other codewords remain unassigned at the start of a compression procedure and we are free to assign them to particular strings as we choose.

In our previous discussions, no limit has been placed on the size of the strings that the dictionary may refer to. There are two reasons why we might want to restrict this. The first is to do with the fact that our dictionary will be held in memory, and we cannot allow the dictionary to grow unpredictably large. Secondly, for reasons that we will come on to, on decompression, we will have to hold entire strings together before outputting the bytes that make them up, and we will want to limit the holding store used to do this. These are practical restrictions, rather than algorithm necessities, but realising the algorithm in tangible form is what we're aiming to do. Now, as each string in a dictionary is one that we have seen in the input data stream, and we only add new entries when we see a string which has not been seen before (but we do always add it) then it must always be the case that a new entry consists of a string that already has an entry in the dictionary, plus a single byte which caused the string to be 'new.' Therefore, we can always make a new dictionary entry by storing a byte value and a pointer to another entry in the dictionary, where the entry pointed to, appended with the byte value, equals the new string being added. If a codeword is 12 bits, and the byte takes 8 bits, all 4096 entries in the dictionary will need 80K bits in total. This, then, gives us a predictable maximum dictionary size which is a manageable quantity.

Now I've gone through some of the practical aspects of implementing this algorithm before embarking on describing the actual algorithm in detail, because the algorithm is not independent of practical considerations. It was designed to be implemented in digital form, and the way it is described is based on these practical concerns. So the next section describes the algorithm in more general terms, but you should see that the method is shaped by the restrictions we have placed on our implementation. We will first look at compression, working through an example to illustrate what the process is. The example we will use is of encoding the 14-byte message:

**RINGADINGDING.**

## Compression

The table below shows the sequence of operations when compressing the example. It is assumed that, at the start of the compression, the dictionary is in its initial state; i.e. it contains entries for the 256 single byte strings only. So as to avoid getting confused with actual numbers, the following convention will be used. All of the bytes will be represented by their ASCII character (e.g. R). A codeword for a string will have its string surrounded with parentheses; e.g. (IN) is the codeword for the string 'IN'. At this point it won't matter what value is assigned to the codewords, so long as it is understood that they will all be different. The table below shows the 14 steps in the algorithm's procedure.

| Input | String | Match? | Build Entry | Output | |
|---|---|---|---|---|---|
| R | R | ✓ | - | - | 1 |
| I | TI | × | (R)I | (R) | 2 |
| N | IN | × | (I)N | (I) | 3 |
| G | NG | × | (N)G | (N) | 4 |
| A | GA | × | (G)A | (G) | 5 |
| D | AD | × | (A)D | (A) | 6 |
| I | DI | × | (D)I | (D) | 7 |
| N | IN | ✓ | - | - | 8 |
| G | ING | × | (IN)G | (IN) | 9 |
| D | GD | × | (G)D | (G) | 10 |
| I | DI | ✓ | - | - | 11 |
| N | DIN | × | (DI)N | (DI) | 12 |
| G | NG | ✓ | - | - | 13 |
| <EOF> Use best match | | | | (NG) | 14 |

From the table above, the first character input is 'R', as shown in the leftmost column. Since this is the first input of any kind, the current string we are working with must be an empty (or NULL) string. Adding the 'R' now makes it a single character string, as shown in the next column. Is this string in the dictionary? Well, yes. All single character strings are always in the dictionary; that is how it is initialised. The third column indicates this with a tick. When a match is found in the dictionary, no new entry is built (the matched entry already exists), and nothing is output (under normal circumstances). The next character is input, 'I', and the current working string becomes "RI". This string is not in the dictionary. We haven't built any new entries yet, so only single character strings can match. The dot in the third column shows that no match is to be found. So, a new string has been seen, and an entry must be built for it.

From an algorithm point of view, the entire string could be stored in any manner, so long as it can be matched against (however inefficiently) at some future point. However, we're working toward a practical implementation, and by introducing a representation (that isn't strictly necessary) now, will save us a lot of trouble later. Since the entry to be built is for a string which must be composed of a string already in the dictionary, plus one character, then all that is needed to store the new string is the codeword for the string already in the dictionary, and the character which, when appended, makes the new string. If it is not clear to you why this works, remember, we keep pulling in characters from the input, appending them to the current string, until the current string is not in the dictionary. Therefore, the current string at that point must be one character longer than a string with a dictionary entry. Thus, shown in column 4 of the table, the new entry is shown as '(R)I', meaning the codeword for the string "R" (a single character string in this case), followed by the character 'I', representing the string "RI". At this point output is required. On a mismatch we output the codeword for the string last matched against. This is the same codeword as was added in the new entry we built (in this case (R) ).

Now, you might ask, why not output the codeword for the new entry we just built, as this represents all the input we have had so far? Well, when we decompress the data, we'd be in the unfortunate position of receiving a codeword for a string which wasn't in the dictionary. For example, (RI) would be the first codeword for decompression which clearly cannot exist in a dictionary which is initialised only for single character strings. As we shall see later, even if we don't output new entry codewords as we build them, this problem is not entirely avoided. More of this later. So (R) is output, which still leaves the input character 'I' unrepresented, and so this becomes our new string, lest we forget about it.

The next input character is 'N', making a new current string of IN. There is no match in the dictionary, and so '(I)N' is added, as for '(R)I' previously. The codeword '(I)' is output, and the new string becomes "N". Now this procedure remains the same for the next four input characters (G, A, D, I) where four new entries are built ((N)G, (G)A, (A)D and (D)I) and four codewords are output ((N), (G), (A), (D)).

Then, something more interesting happens (at last!). We have "I" as our unprocessed string, and 'N' as our input character, giving us "IN", which is (hurrah!) in the dictionary, built at row 3. Now all we have to do is remember the entry for "IN" (stored as (I)N) and input the next character. No codeword is output, and no new entry is built. The next character input is 'G', giving us "ING" as our current string, and this is not in the dictionary. In fact, no three-character entries have yet been built. The procedure for a failed match applies once more. A new entry is built, stored as

(IN)G. That is, the entry number for (IN) is stored, along with the character value for 'G'. The string we have matched is output (i.e. (IN)); the first dynamically generated entry codeword. And the current string is set to "G".

The rest of the input is processed in the same manner until the end. There is one further match to a dynamically generated dictionary entry, and it is processed in the same manner as described above. The only thing left to clear up is what to do with the end of the data stream. It will always be the case that at the end of processing there will be a string that is matched in the dictionary but has not been represented in the output. This could be a single character string, or, as in our example a match with a dynamically allocated entry. Since the normal procedure is to carry it forward for appending the next input character, we must do something different, since no more input is forthcoming. In this case we simply 'pretend' that there is no match and output the codeword for the current outstanding string. No entry is built, and the current string is effectively set to the null string. All the input has been represented in the output in an encoded form which must, therefore, contain enough information about the original data to recover it. It doesn't explicitly contain the dictionary, but, as we shall soon see, this too has been encoded in the output data; a feature which is one of the major attractions to this class of algorithm. So, without further ado, let's decompress the encoded data, and see what happens.

## Decompression

In decompression, the input must consist of the codewords we have previously generated as the compressed output, in the order in which we generated them. These will be, as in our compression example, a mixture of root codewords and dynamically allocated codewords. We can only start with an initialised dictionary, as the final compression dictionary has not been stored with the data. This restricts the point at which decompression can begin within a compressed data set. Only at points where the dictionary was in its initial state during compression can decompression begin. In the simple examples here, this means at the beginning of the data.

The table below shows the decompression sequence for our previously compressed example. The first six codewords are all root codewords, and their byte values may be sent directly to the output. At each stage (except for the first codeword) a dictionary entry is built with a byte value of the current input codeword, and a pointer to the previously processed codeword's dictionary entry. I.e. an entry representing the last string processed with the current codeword's byte value appended to it. Hence the

first entry built is (R)I, since, when (I) is input, the last codewords processed was (R) and the byte value of (I) is 'I'.

| I/P | Byte Value | Ptr | Root? | LIFO | Build Entry | O/P |
|---|---|---|---|---|---|---|
| (R) | R | - | ✓ | R | | R |
| (I) | I | - | ✓ | I | (R)I | I |
| (N) | N | - | ✓ | N | (I)N | N |
| (G) | G | - | ✓ | G | (N)G | G |
| (A) | A | - | ✓ | A | (G)A | A |
| (D) | D | - | ✓ | D | (A)D | D |
| (IN) | N | (I) | × | N | | |
| | I | - | | NI | (D)I | I |
| | | | | | | N |
| (G) | G | - | ✓ | G | (IN)G | G |
| (DI) | I | (D) | × | I | | |
| | D | | ✓ | ID | (G)D | D |
| | | | | | | I |
| (NG) | G | (N) | × | G | | |
| | N | - | ✓ | GN | (DI)N | N |
| | | | | | | G |

The seventh codeword input is (IN); a non-root codeword. It already has an entry in the dictionary at this point, as show on the third line in the figure. This entry has a byte value of 'N', and so this is placed on a 'LIFO'. LIFO stands for 'last in first out'. As we follow this example through, you will see that the byte values recovered in a multi-byte codeword are in reverse order. Therefore we must store them until the entire string is decoded, and then output them in the correct order. A 'LIFO' is used, where we can 'push' bytes into it, and then 'pop' them when we have all the bytes- the last byte pushed being the first byte popped.

The entry for (IN) also has a pointer to another entry; in this case it is the root entry (I). Its byte value is also pushed onto the LIFO and, as it is a root codeword, the contents of the LIFO are flushed to the output. At this time, an entry is built using the same rules as before. The new entry has a pointer to the previously processed codeword (i.e. (D)) and a byte value equal to the current codeword's last pushed byte value. Since we terminated the link list traversal pointing to (I), the byte value is 'I', as shown in the figure. The rest of the input codewords are processed in a similar manner, with the entire original data recovered, and the dictionary left in the same state as at the end of compression.

So, we have managed to recover the data, without loss of information, and without storing the dictionary—all was encoded uniquely within the compressed data itself. And that would be all there is to it, except there is one slight twist in the tale. There is a circumstance where, during decompression, a codeword is encountered which has no entry in the dictionary! This is the KωK exception case.

## The KωK Exception Case

Welch describes in his paper a particular corner case in the LZW algorithm which, if processed as described above, will cause a breakdown of the procedure. It is called the KωK (kay-omega-kay) exception case, since inputting the sequence KωK, where K is any single character and ω is a string for which there is a dictionary entry, causes a codeword for the string KωK to be output (during compression) at the same time as it is built in the dictionary. This causes a problem on decompression since no entry will have been built on encountering it, as entries are built after the codeword is processed. More properly, the sequence KωKωK causes the problem, and the clue to its solution is in the fact that the entry needed is the very next entry we would have built at the end of processing the codeword causing the problem. So, we might have enough information to construct the missing entry uniquely and recover the situation.

To see how, an example is in order. The simplest KωK sequence is the repeated character. This is because K can be any character, and all single character strings have entries in the dictionary by default. The table below shows a repeated character compression sequence.

| I/P | String | Match? | Build Entry | O/P |
|---|---|---|---|---|
| K | K | ✓ | | |
| K | KK | × | (K)K | (K) |
| K | KK | ✓ | | |
| K | KKK | × | (KK)K | (KK) |
| K | KK | ✓ | | |
| K | KKK | ✓ | | |
| K | KKKK | × | (KKK)K | (KKK) |

The procedure followed is, I hope you can see, exactly that in the previous compression example. Entries are built when no match is found, and the last character input is carried forward as the new current string. Now, if we are presented with the codeword sequence shown in the output column of the figure, what will happen? We can process the first codeword as it is a root codeword with a known byte value. No entry can be built, as no previous codeword has been processed. The

next codeword is (KK). But this is a dynamically allocated codeword, and we haven't built any entries yet. There are a couple of slightly different methods for detecting, and then correcting this problem, but, basically, the problem is detected if the value of the codeword is the same as the next dictionary entry that will be built (and they are built in ascending order). It is corrected by placing the byte value of the last processed codeword onto the LIFO and re-submitting the last codeword value again in place of the current problematic one. This will, once the FIFO is flushed, place the right characters on the output, and will result in the 'missing' dictionary entry being built, using the normal procedures. An alternative exists where the missing entry is built first, but the result is just the same, and it is left to the reader to figure out the details. This final detection and correction works for all KωK cases, and ties up the last lose end, as far as the basic algorithm goes.

Describing an algorithm and manipulating symbols on paper shows how an algorithm might work, but we haven't arrived yet at a method which is fully practical. There are still some details to consider before the above can become a working piece of software, or a logic implementation. Some additional (fortunately much simpler) algorithms need to be roped in, and some design constraints and limits placed on the algorithm to map the symbolic representation onto more practical units of information, such as bytes. But that must wait for the next part.

# Part 3: LZW Implementation

## Introduction

In the previous part we looked at the LZ class of lossless data compression algorithms and focussed on the LZW algorithm in particular. Now it's time to put our money where our mouth is and actually make a practical implementation. So, in this third part of the document, as a working example, we'll look at implementing a simple LZW algorithm (slzw). In doing this we can highlight the limitations and boundaries that exist when turning an algorithm into a working implementation. LZW is chosen as it follows naturally on from the previous part, and because it forms the basis of many practical implementations, including GIF graphics files, hardware compression in mass storage and many other applications.

In the previous part, the LZW algorithm was discussed in general terms and idealistically. In order to realise a practical implementation, whether logic or software, some boundaries must be defined and choices made. In the next few sections a simple software implementation is described, laying out where choices are made, and what alternatives might exist, as well as alluding to the architecture of a logic solution. It may seem that the discussion is ordered the 'wrong' way round, in that we'll look at the actual compression and decompression code last, but this is because those functions are built on the features of the ancillary functions. The compressor matches and builds entries in the dictionary and sends on codewords to the packer. The decompressor gets codewords from the unpacker and reads and builds entries in the dictionary. So we need to understand these functions first. So let's start by looking at the dictionary.

## Dictionary

In the LZW discussion of the previous part, there was an unspoken assumption that is actually impractical—the dictionary is infinitely large. In this section we are going to look at some bounding of the dictionary to make it practical, and how a dictionary is realistically implemented. The goals are twofold. Firstly, what sensible limit on dictionary size is acceptable, and secondly, how can we construct it so that it is reasonably fast to use.

## Bounding the Dictionary

The actual size (in bytes) of a dictionary is determined by two things: the size of each entry, and the number of entries. Let's bound the latter of these first (why not!). Making the dictionary as large as possible more nearly approximates the ideal of infinite, but you'll need much more space to store it, and (perhaps) a very long time to search it for an entry. Making it very small (and it *must* have at least 256 + 1 entries, or we're not making a compressor!) is not likely to yield a useful implementation. A sensible approach is to choose a power of 2 size from $2^9$ upwards. Each increment yields a dictionary which is double in size, and requires double the resources, so this quickly becomes a limiting factor. In all practical LZW based implementations I have seen (both software and hardware) dictionaries are between 1K and 64K bytes. Any smaller and compression figures are severely hit; any larger, and the gain in performance is negated by the resources and search times required. So we will choose a midway point of 4K, which is the size used in the DCLZ algorithm (used in archive storage devices and others). By making this choice, we have significantly bounded the dictionary. It has 4096 entries, and this limits the maximum codeword width to 12 bits. Knowing this defines the entry width to 12 + 8 = 20 bits, as the dictionary entry pointers are the same size as the codewords (indeed they are synonymous). As a minimum then, we need (4096 × 20)/8 = 10K bytes of resource for a dictionary.

## Dictionary Resetting Choices

Now we've bound our dictionary, there is the problem of what to do when it's full. This question opens up a whole can of worms as there many possible choices which may have some advantages for some cases, whilst being quite disadvantageous in others. The issue is, when the dictionary is full, how relevant is it to current input data? If the data remains similar in character (say we're compressing a large source of English text), then it may be very relevant. On the other hand, if we're compressing, say, an archive file which contains an eclectic mix of small files of different types, then the dictionary will become irrelevant when the file type switches after it's become full. The simplest thing to do is reset the dictionary when its full (or at least when we want to build a new entry after its full). This is great, but there is a penalty in that compression ratios suffer after a reset when there are less entries to match against. An advantage is that the compression and decompression resets don't have to be explicitly synchronised, since the dictionary is built up in the same manner in both cases, and reaches full in the same place (well, almost—see below). A more elaborate method is to somehow measure the 'effectiveness' of the dictionary, and only reset it when it becomes 'ineffective'. How to do this is the subject of much research and

patent filing, and without going into details, measuring the compression ratio over a set number of input bytes can achieve this. An even more complex enhancement is to only reset *some* of the entries which are less 'relevant'. But that is beyond the scope of this document. Suffice it to say that the common factor with these methods is that the resetting of the dictionary is done externally to it. Therefore we need to synchronise between compression resets and decompression resets. This is usually done by inserting a marker in the compression stream. Reserving one of the codewords used for a dictionary entry is one method. In DCLZ this is the Reset codeword (value 0x001). Whenever the dictionary is reset, the compressor inserts a Reset codeword. On decompression, if a Reset is encountered then the dictionary is reset before continuing. The advantage of the special codeword is that decompression is immune to the method used to determine reset on compression. So it can be made completely configurable, dynamic, or whatever, during compression without precluding the decompressor from decoding the data—even a different implementation by a third party. Having said all that, we're going straight back to square one, and choosing to implement a 'reset on full' approach.

## Implementing a Dictionary

We may have defined a dictionary's boundaries, but this doesn't tell us how to use it. There are two main things we must be able to do with the dictionary; add new entries and determine whether a given value is in the dictionary (is 'matched'). Building an entry is a trifle more involved than simply writing a value (but not much more). The address used when writing the entry is managed by the dictionary itself. An incrementing counter is kept, initially set to the lowest dynamic entry, and incremented as each new entry is written. This counter is managed by the dictionary but is going to be needed externally and so must be exported (preferably behind a 'data hiding function', but our example won't go this far for the sake of clarity). An example function is shown below. Note that I am using a sparse curly-brackets style for the sake of clarity and fragment size, just as in Kernigan and Ritchie's "*The C Programming Language*" (so do not complain):

```
static unsigned int build_entry(codeword_type codeword, byte_type byte,
                                boolean compress_mode) {
    static unsigned int codeword_len = MINCWLEN;

    if (next_available_codeword == DICTFULL) {
        next_available_codeword = FIRSTCW;
        codeword_len            = MINCWLEN;
        return codeword_len;
    }

    dictionary[next_available_codeword].pointer = codeword;
    dictionary[next_available_codeword].byte    = byte;

    if (compress_mode)
        indirection_table[codeword][byte] = next_available_codeword;

    if (codeword_len < MAXCWLEN) {
        if (next_available_codeword == (1U << codeword_len) -
                                (!compress_mode ? 1U : 0U))
            codeword_len++;
    } else if (!compress_mode && next_available_codeword == (DICTFULL-1))
        codeword_len = MINCWLEN;

    ++next_available_codeword;
    return codeword_len;
}
```

This code assumes a pre-existing global array (dictionary) each entry of which is a structure with a pointer and byte field. A codeword and byte to be written is passed in, and a flag to indicate whether we're compressing (as opposed to decompressing—the dictionary routines are common to both). The first lines test for a full dictionary and reset it if it is, before returning. The entry to build in that case is effectively discarded. If the dictionary isn't full the codeword and byte values are written to the dictionary at the address indicated by the counter `next_available_codeword`, which is incremented before returning. Before returning, though, some additional processing is done. An indirection table is updated. More on this below, but it is only a method of quickly locating an entry during a match search, which could be done in many different ways, such as a hash table or even a linear search if speed is not important. The next bit of code is determining whether the current dictionary entry (`next_available_codeword`) is larger than the current packing/unpacking width (see "Packing and Unpacking" below). If it is, `codeword_len` is incremented to increase the packing/unpacking width. Finally, when decompressing, the entries being built lag behind compression by one codeword (the famous KωK exception case is caused by this), so we have to reset the codeword_len one entry early to compensate.

Searching the dictionary to see whether a given data value matches one already built is more difficult. As mentioned above, one solution is to linearly search from the first dictionary entry to the last built entry, but this is *very* slow. A hash table is a much

better idea. For those unfamiliar with this technique, basically instead of storing the entry linearly against an address, an address is generated from the data, trying to make it pseudo-random like. So usually, bits are jumbled up, and/or inverted etc. At this data dependant location is stored the 'codeword' (`next_available_codeword`) that would have been the address of the linear implementation, along with the data itself (or some unique part of it). When searching for an entry, the hashed address can be regenerated using the same algorithm, and that entry inspected for the data we're try to match. If it's the same, then the stored codeword is retrieved. This sounds great, and is very fast, but has some drawbacks. For an effective hashing algorithm, the number of entries must be greater than for a linear dictionary. This is because of another drawback—different values can generate the same address (whatever hashing algorithm you come up with). This is alleviated somewhat by making the hashing dictionary bigger (at the cost of additional resources) and making 'collisions' less likely, but even so, these collisions must be dealt with, and extra complexity is added to detect a collision and end up at a unique location for a given entry. This method can be used, however, in both software and hardware implementations affectively, and some early DCLZ logic implementations used this method. More recently hardware implementations use a type of memory called content-addressable-memory (CAM). CAMs can act like ordinary SRAM, but in addition allow a parallel search of a value in all its locations at once, returning a 'match' status and an address of the matched data (if any). We can't do this same technique in software, but we can avoid the hashing penalties at the expense of more memory requirements—but it is much more plentiful on a computer than on a silicon chip anyway.

The technique makes use of an indirection table, which is nothing new, but its use with respect to an LZW based dictionary is novel to me as far as I know. We can take advantage of the fact that each entry in the dictionary is unique (if the dictionary is a valid one), and that it is contiguous from the first entry to the top; i.e. has a valid entry, without holes, up to `next_available_codeword`. The table is a two-dimensional array which is 256 wide by 'dictionary size' long (4K in our case). It could have been simply a linear 256×4K array, but the important point is that there is an entry for every single codeword/byte pair that can exist. When a new entry is built in the dictionary, the address of the new entry is stored in the indirection table at `indirection_table[codeword][byte]`; i.e. the unique location defined by the data. This is what's happening in the `build_entry()` function above. When we want to match a particular value (a codeword and byte pair) in the dictionary, the dictionary address is fetched from the indirection table and the dictionary entry compared with the match data. If it is the same then we have a match, and the address from the

table is the codeword we need, or we haven't matched, and the data is not in the dictionary. Let's look at the code:

```
static codeword_type entry_match(codeword_type pointer, byte_type byte)
{
    codeword_type addr;

    addr = indirection_table[pointer][byte];
    if (addr < FIRSTCW || addr >= next_available_codeword)
        return NOMATCH;

    if ((dictionary[addr].byte == byte) &&
        (dictionary[addr].pointer == pointer))
        return addr;

    return NOMATCH;
}
```

Hopefully it is clear that the above code implements the described operation. The only extra bit is a check that the retrieved address is within the valid dictionary range and returning a `NOMATCH` if it isn't. By doing this we avoid having to initialise the table (which can be quite large), and random invalid values cause a mismatch. Of course a random value may lie within a valid dictionary location, but this still doesn't matter. If the codeword/byte pair has never been written in the dictionary, then that random value *can't* contain the value, since all dictionary entries are unique, and we mismatch. If the value has been written in the dictionary, then the indirection table won't have a random value in it at that location, and we get a match.

These two functions constitute the two operations needed by the compression and decompression code and, as stated before, the `next_available_codeword` counter, defining the top of the valid dictionary, is exported (in our case a simple global) to enable inspection of the reset state. One last operation is needed from the dictionary, and that is to read a particular location in decompression, where incoming codewords define this address. In the example code this is achieved by making the dictionary global and reading it directly. Again this is purely for clarity—keeping the number of lines of code to a minimum is essential in a document, but ideally the dictionary would have a data hiding function which returned the pointer and byte value of a given entry, leaving the dictionary array completely local to the dictionary routines.

So we have a dictionary. Before moving on to compression and decompression we still need some more functions. To get the most performance from our implementation, the steady stream of codewords need to be 'packed' as tightly as

possible, and subsequently 'unpacked' before decompressing. We'll look at this in the next section.

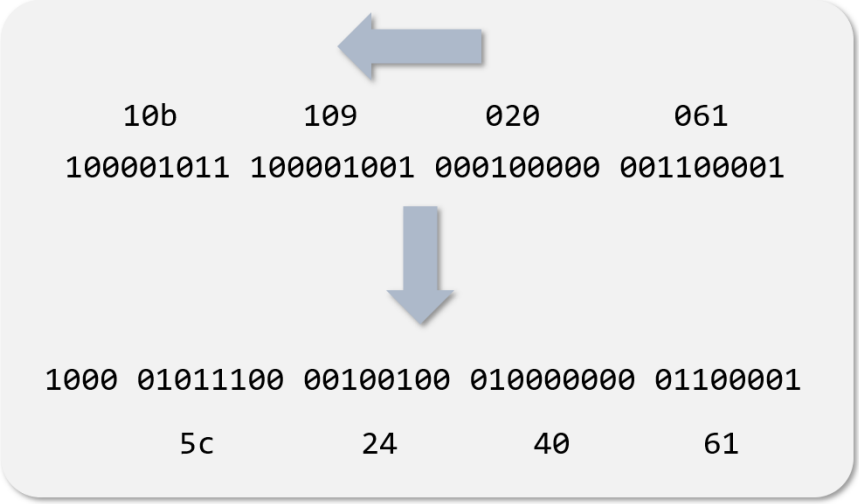
# Packing and Unpacking

The pure LZW algorithm discussed in the previous part will generate a stream of codewords. By bounding the size of the dictionary (e.g. to 4K bytes) that stream of codewords will be fixed (to 12 bits for a 4K dictionary). Since the codewords might not exactly fit into bytes, it is useful to 'pack' the codewords so as to avoid wasted padding bits to make the codewords align to byte boundaries. Packing simply means appending the next codeword at the bit position after where the current output codeword finishes. Unpacking is simply the reverse. Since the codeword size is known, enough bytes are read to have a whole codeword, which can then be passed into the decompression processing. Any remaining bits are held over and used as the beginning of the next codeword.

## Dynamic Packing

The above scheme is all very well, and it avoids the insertion of redundant bits (not good in a compression algorithm). However, from the previous part and from the sections above, you may have noticed that initially, when only a few dictionary entries have been built, all the codeword values are small. As the dictionary is filled, higher value codewords become possible, and increase in bit width up to 12 bits. So, when packing, we can take advantage of this and pack the codewords more compactly by treating them as being less wide than the maximum width. For example, for a 4K dictionary with 12-bit codewords, initially 9 bits are all that are required to pack until the dictionary has an entry in `0x200` (512) when 10 bits are needed, and so on until entry number `0x800` (2048) is built, when 12 bits (the maximum) are needed. There's a choice about whether to pack at the new width when the entry is built, or when a wider codeword must be packed (since there may be a delay between the two). We'll discuss this later but suffice it to say that we can inform the packer what the current width required is and adjust the packing accordingly. This gives us the best packing of codewords we can achieve.

One other arbitrary choice we have is whether to pack codewords least- or most-significant byte first. It doesn't matter which, but we have to choose the same between packing and unpacking! I prefer least significant first, and this is also used in the DCLZ format. But another algorithm ALDC uses most-significant. Using LSB, let's have an example. Suppose we're packing 9-bit codewords:




In the above sequence, time is flowing from right to left, so that the codeword sequence (in hex) is `061`, `020`, `109`, `10b`, with the binary values underneath. Hopefully you can see that the second row of binary values are the same bit sequence as the first but have been split at every eighth bit instead of ninth. Converting back to hexadecimal values now gives a byte sequence of `61`, `40`, `24`, `5c`. We have 4 bits left over, which will form part of the next byte, or if this is the last byte, would be flushed to a byte boundary to give `08`. Unpacking is the reverse, with the bytes read and the correct number of bits for a whole codeword pulled off to form a stream of unpacked codewords.

Now to determine when to update the packing width. The simplest approach is to set the width to match the last built dictionary. Since we can only output codewords that are built in the dictionary, we'll never find ourselves in the position of trying to pack, say, a 12-bit codeword when we're only packing at 11 bits. It is also easy to synchronise between packing and unpacking as the dictionary is built in the same manner between compression and decompression. There are two issues with this however; a small and a not so small one. The small issue is that the packing/unpacking has to have access to some internal dictionary state; i.e. the highest built entry. Not a disaster, but it would be better if these mostly disassociated functions remained decoupled.

The second more relevant issue is that when an entry is built at one of the bit boundaries, it *cannot* be used straight away. At least one (necessarily smaller) codeword will be output first, and more likely several codewords. These codewords will be packed more loosely than is strictly necessary, wasting bits. We can solve this by waiting to up the packing width until a codeword is ready to be output that requires the wider width. The packer can determine this by inspecting the upper bits of the codewords and if any are set higher than the current packing width, it can

increase as necessary. There is also no need to inspect any dictionary state, and its input is all that's required. Unfortunately, during unpacking it is not possible to know in advance when the packer would have increased the width. So some sort of indicator must be inserted into the data stream to inform the unpacker to increases its width. This is usually done by reserving one of the dictionary codewords to have a special meaning. It makes sense if we choose one of the lowest numbered codewords (e.g. `0x100`) as this will be nine bits, and so most efficient. In DCLZ, this codeword is called a 'Grow' codeword, and (for strange historical reasons I won't go into) has a value `0x002`. Whenever the packer receives a codeword which is too big, it inserts a Grow codeword (at the current packing width) and increments its current width. When the unpacker receives a Grow codeword, it increments its current width, and then discards the codeword as the decompressor knows nothing of this type of command. It should be noted that it is possible to jump more than one width (from say 10 to 12), and the packer must insert as many Grow codewords as necessary to get to the required width. The advantage of this method is that it decouples the dictionary from packing and doesn't waste bits in the packed codewords. However, bits are wasted in outputting a Grow codeword, and a dictionary entry has been wasted to reserve a number for the Grow. In addition it is more complex to implement and the benefit may be dubious. Although there $may$ be a significant lag between a dictionary entry being built and its codeword being output, usually this is not the case as much repetition is localised in common data (though there are no hard and fast rules on this). My own feeling is that it is an optimisation too far, and that dictionary synchronised packing width is simpler and of no significant degradation in performance.

One final thing before we look at some code implementation. If the dictionary ever gets reset, the packing/unpacking width must also revert to its minimum value. This may mean that the packer/unpacker must interpret another special codeword (Reset) or have further tendrils into the dictionary to determine when it is reset.

## Implementation

Let's see some code for a packer:

```
static void pack(codeword_type ip_codeword, unsigned int codelen,
                 boolean flush, FILE *ofp)
{
    static unsigned long barrel  = 0;
    static int           residue = 0;

    barrel  |= ((ip_codeword & CODEWORDMASK) << residue);
    residue += codelen;

    while (residue >= (flush ? BITSIZE : BYTESIZE)) {
        putc((barrel & BYTEMASK), ofp);
        barrel >>= BYTESIZE;
        residue -= BYTESIZE;
    }
}
```

This function has four inputs. The codewords arrive as `ip_codeword`, and the packing width has been determined from dictionary externally and is input as `codelen`. The flush flag indicates the end of data case mentioned in the above example, where the final bits must be padded to a byte boundary. Finally `*ofp` is simply a file pointer to the destination output stream.

The code works by appending the `ip_codeword` onto `barrel` (a barrel shifter) above any bits that remain on the shifter, as indicated by `residue`. For every `ip_codeword` added, `residue` is incremented by the current width (`codelen`). Then, for as many whole bytes as there are on the barrel shifter (as indicated by `residue`), the bottom eight bits are output to the output file, the barrel shifted right 8 bits and residue decremented by 8. The exception to this is if `flush` is set, when bytes are output until there are no residue bits (`residue` is 0).

An important point to note is that the maximum number of bits on the barrel shifter is never more than 7 plus the maximum codeword width (e.g. 19 for a 4K dictionary). So the type of the barrel in a C program (or width of a register vector in a hardware implementation) needs to take this into consideration. Unpacking is the reverse of packing.

```
static int unpack(codeword_type *codeword, unsigned int codelen, FILE *ifp)
{
    static unsigned int residue = 0, barrel = 0;
    short int           ipbyte;

    while (residue < codelen) {
        if ((ipbyte = getc(ifp)) == EOF)
            return EOF;
        barrel  |= (ipbyte & BYTEMASK) << residue;
        residue += BYTESIZE;
    }

    *codeword = (barrel & ((1 << codelen) - 1));
    residue  -= codelen;
    barrel  >>= codelen;

    return ipbyte;
}
```

The arguments are similar to `pack()`, but `*codeword` is a pointer where `unpack()` will output a codeword. The `codelen` input is the same, and `*ifp` is the input stream file pointer. So here, the unpacker appends bytes to the barrel shifter, adding 8 to `residue`, until enough bits for a codeword are read. The bottom `codelen` bits are then returned in `*codeword`, the `barrel` shifted right to delete those bits, and `residue` adjusted accordingly. One special case is when end-of-file is reached (`EOF`). Here no processing is done, but `EOF` is returned to tell the decompressor that no more data is coming. So, normally we must return something other than `EOF`, which what the last line is about.

So now, with a dictionary and packer/unpacker, we have enough to look at compression and decompression (finally!). This is discussed in the next section.

# Compression

Having (wisely) defined the dictionary and packing functions, compression becomes a breeze. We simply keep pulling bytes in, matching the byte and the previously matched codeword (if any) in the dictionary until there is no match. Then we output the most recent matched codeword and build a new entry for this codeword and byte that we mismatched. The code is shown below:

```
static void compress(FILE *ifp, FILE *ofp)
{
    codeword_type previous_codeword = NULLCW;
    unsigned int code_size          = MINCWLEN;

    codeword_type match_addr;
    int           ipbyte;

    while ((ipbyte = getc(ifp)) != EOF)
        if (previous_codeword == NULLCW)
            previous_codeword = ipbyte;
        else
            if ((match_addr = entry_match(previous_codeword, ipbyte)) == NOMATCH){
                pack(previous_codeword, code_size, FALSE, ofp);
                code_size = build_entry(previous_codeword, ipbyte, TRUE);
                previous_codeword = ipbyte;

            } else
                previous_codeword = match_addr;

    pack(previous_codeword, code_size, TRUE, ofp);
}
```

So, the `compress()` function has two arguments, which are the input and output file pointers. The heart of the function is the while loop pulling in bytes into `ipbyte`, until the end-of-file. The special case of the very first byte is caught when `previous_codeword` is `NULL`. The `previous_codeword` variable holds the best match we have so far, but at the beginning we haven't done a match. So in this case the `ipbyte` value is simply copied to `previous_codeword`, as we now have an implied match on the 'root' codeword for the input byte. In our scheme root codewords run from 0 to 255 matching the bytes—a sensible choice I trust you'll agree. However, not all schemes follow this obvious choice and DCLZ, for example, has root codewords which run from 8 to 263! (It has 8 reserved codewords from 0 to 7).

Normally, we pull a byte in and do a match by calling the dictionary function `entry_match()`. If we have a match, then the returned match address becomes the `previous_codeword`, and we go round the loop again. If we didn't match, then the best match we had so far is output by sending to the packer (see the "Packing and Unpacking" section above) and a new entry built for the failing match. We have output `previous_codeword`, but the byte in `ipbyte` is not yet represented in the output, so we must copy it to `previous_codeword` as it is now our best match (or its root codeword value is, which is the same thing for us). When end-of-file is reached, and we drop out of the loop, the output packer is called once more, as we

will have a best match still in `previous_codeword`, even if it is just the last byte's root codeword. The `pack()` function is normally called with its third argument as `FALSE`, but the last call has it set to `TRUE`. This informs the packer that no more data is coming and to 'flush' to a byte boundary. Also the `code_size` variable defines the 'packing width' to use. It is based on the highest dictionary entry built and is set when `build_entry()` is called which returns the decoded value.

# Decompression

Decompression is only slightly more involved than compression. Here we pull in a stream of codewords and traverse the dictionary stuffing the byte values of the entries onto a stack. When a root codeword is reached, the stack is flushed to the output, reversing the order of the bytes read from the dictionary. Let's look at some code:

```
static void decompress(FILE *ifp, FILE *ofp) {
    static byte_type    byte, stack[MAXDICTSIZE];
    static unsigned int stack_pointer = 0;
    codeword_type       pointer, ip_codeword, prev_codeword = NULLCW;
    unsigned int        code_size = MINCWLEN;

    while (unpack(&ip_codeword, code_size, ifp) != EOF) {
        if (ip_codeword <= next_available_codeword) {
            pointer = ip_codeword;
            while (pointer != NULLCW) {
                if (pointer >= FIRSTCW) {
                    if (pointer == next_available_codeword &&  (prev_codeword != NULLCW))
                        pointer = prev_codeword;
                    else {
                        byte    = dictionary[pointer].byte;
                        pointer = dictionary[pointer].pointer;
                    }
                } else {
                    byte    = pointer;
                    pointer = NULLCW;
                }
                stack[stack_pointer++] = byte;
            }

            while (stack_pointer != 0)
                putc(stack[--stack_pointer], ofp);

            if (prev_codeword != NULLCW)
                code_size = build_entry(prev_codeword, byte, FALSE);
            prev_codeword = ip_codeword;
        } else {
            fprintf(stderr, "***Error---UNKNOWN CODEWORD (0x%03x)\n", ip_codeword);
            exit(DECOMPRESSION_ERROR);
        }
    }
}
```

Like compression, a 'while' loop forms the heart of the function. Codewords are fetched by calling the unpacker (`unpack()`) which returns a codeword into `ip_codeword`. A check that the codeword is valid is done before proceeding (it can happen!) and an error printed if something's wrong. Otherwise we traverse the dictionary, which is what the inner 'while' loop is doing. When the pointer (which holds the current dictionary address whilst we're traversing) is a root codeword (i.e. less than `FIRSTCW`) then we construct a root codeword, making the byte value simply the pointer (in our example) and setting `prev_codeword` to `NULL`. If it's a dynamic codeword then we get the new byte and pointer value directly from the dictionary. However, we have to detect and deal with the KωK case. This is detected when the pointer is equal to the dictionary's `next_available_codeword` value (and this is not the first time through the loop—`prev_codeword` is not `NULL`). In that case we reconstruct the pointer from the `prev_codeword` of the last time round the loop, and the byte value is also correct from last time (i.e. the last output byte that terminated the previous string). In all cases, the byte value we determined gets added to the stack. When we reached a root codeword, the inner loop terminates and the stack gets flushed to the output, reversing the bytes. A new entry in the dictionary is then built (except first time round) using the `prev_codeword` value and the last byte pushed on the stack. The input codeword we just processed then becomes the previous codeword for the next loop.

As in the packer during compression, the unpacker is informed of the current codeword width based on the dictionary's highest entry, and `code_size` is updated on every call to the dictionary routine `build_entry()`.

# Conclusions

So that's pretty much all the code we need. A top level `main()` function can define the input and output streams (`*ifp` and `*ofp`) and select between compression and decompression. Actually I have done this for you and the entire code can be accessed on github. This is a single, self-contained file (`slzw.c`) which should compile on just about any platform (I've tried both Linux and Windows—you'll need `getopt()` for windows though; try the link here), and has all the functions already mentioned and everything else that's required. The whole thing is less than 250 lines of code, and still achieves a satisfactory performance in both speed and compression ratio. Its usage is straight forward, with just a few command line options:

```
Usage: .\slzw.exe [-h] [-d] -i <filename> -o <filename>

Options:
   -h Print help message
   -d Perform decompression
   -i Input filename
   -o Output filename

All debug information sent to standard error
```

The input file to process is specified with the `-i` option, and the output file is specified with the `-o` option. By default the program compresses from input file to output file. For decompression, the `-d` option is used.

Feel free to try enhancing the code for larger dictionaries, different reset strategies, or any other enhancement you can think of. The code is open source and the patent for the LZW algorithm, originally held by Unisys, has now expired in all countries. So you are also free to incorporate this code into your own products without infringing any patents.

We have now finished with this review of lossless algorithms and will move onto looking at lossy algorithms by discussing JPEG and JFIF concepts and format in the next part.

# Part 4: JPEG

## Introduction

In the previous parts of the document, I looked at some lossless data compression algorithms, with an implementation (in C) of the LZW algorithm as a case study. In this fourth part we will look at a lossy algorithm for images; JPEG. This will use some of the topics which we have already discussed, as a run-length encoding and Huffman encoding is used in the JPEG format. So if you have not read the previous parts, then I recommend reading at least part 1, before embarking on this part on JPEG.

JPEG format files are now ubiquitous on the Internet, in digital cameras and on smart 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 other more advanced techniques.

In this part of the document a discussion of the concepts of the JPEG (and JFIF) format are presented in the context of going on to describe a decoder software implementation which is, itself, a jumping off point for a potential logic 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. This part of the document consists of three sections. The first introduces the basic concepts of JPEG and narrows this down to details relevant to implementing a JPEG decoder. This part is meant to be read *before* tackling the later implementation part unless familiarity of the format is already mastered. The following part details the internal architecture and design of both the software implementation of a fully working JPEG decoder, with accompanying source code available for download.

## 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 discussion. 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 × 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, with different parameters. 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 + 1.722 (Cb-128) |

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.

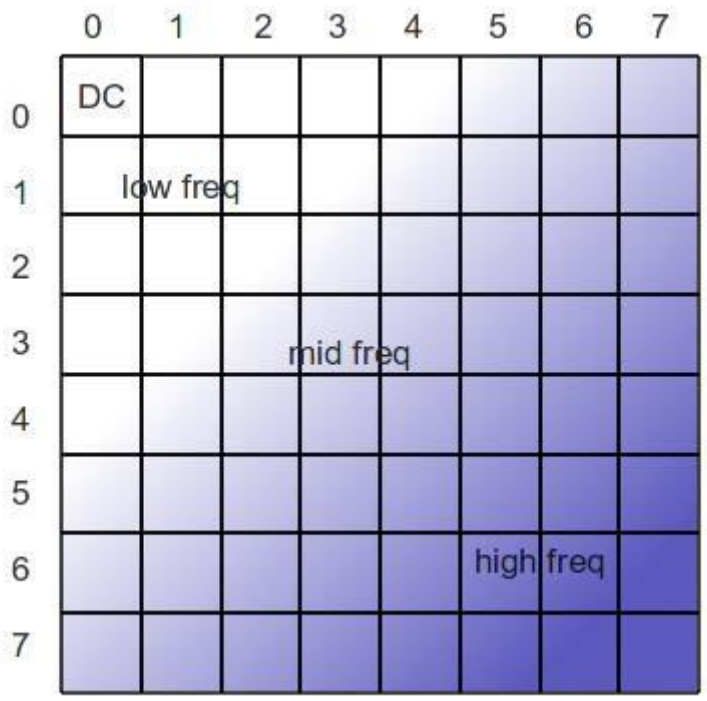
## DCT and iDCT

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 checkerboard 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 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 the figure 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.




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 the equations below.

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

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

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

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 constant 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 part 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 document.

## 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 the figure below:

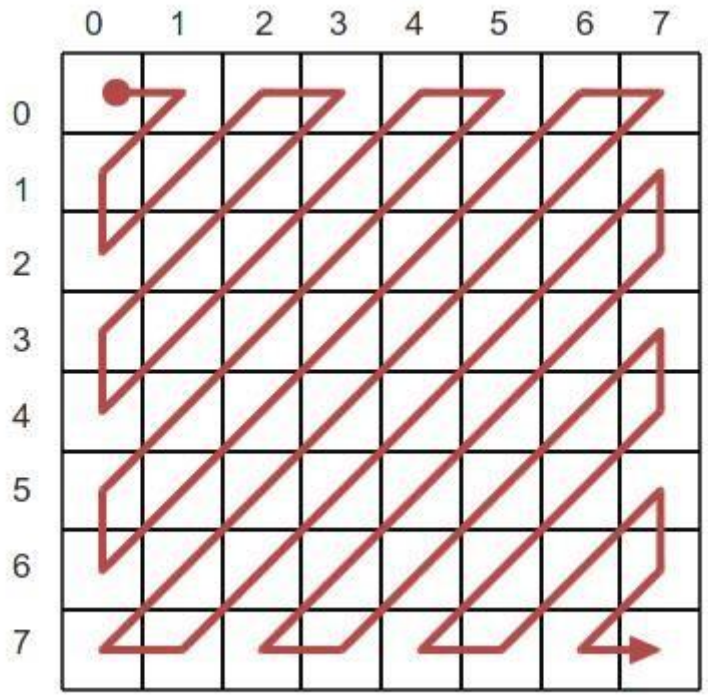
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 16 | 11 | 10 | 16 | 24 | 40 | 51 | 61 |
| 12 | 12 | 14 | 19 | 26 | 58 | 60 | 55 |
| 14 | 13 | 16 | 24 | 40 | 57 | 69 | 56 |
| 14 | 17 | 22 | 29 | 51 | 87 | 80 | 62 |
| 18 | 22 | 37 | 56 | 68 | 109 | 103 | 77 |
| 24 | 35 | 55 | 64 | 81 | 104 | 113 | 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 |

The above two examples are lifted straight from the JPEG specification, with the top intended for luminance data, and the bottom 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, dependent 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. The figure below shows the order in which we serialise the two-dimensional data.




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 the figure at the beginning of this section, 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 using a type of run-length-encoding discussed in the first part of this document. The encoding is somewhat different than the simple example I gave previously to explain the concepts.

| 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 |

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.

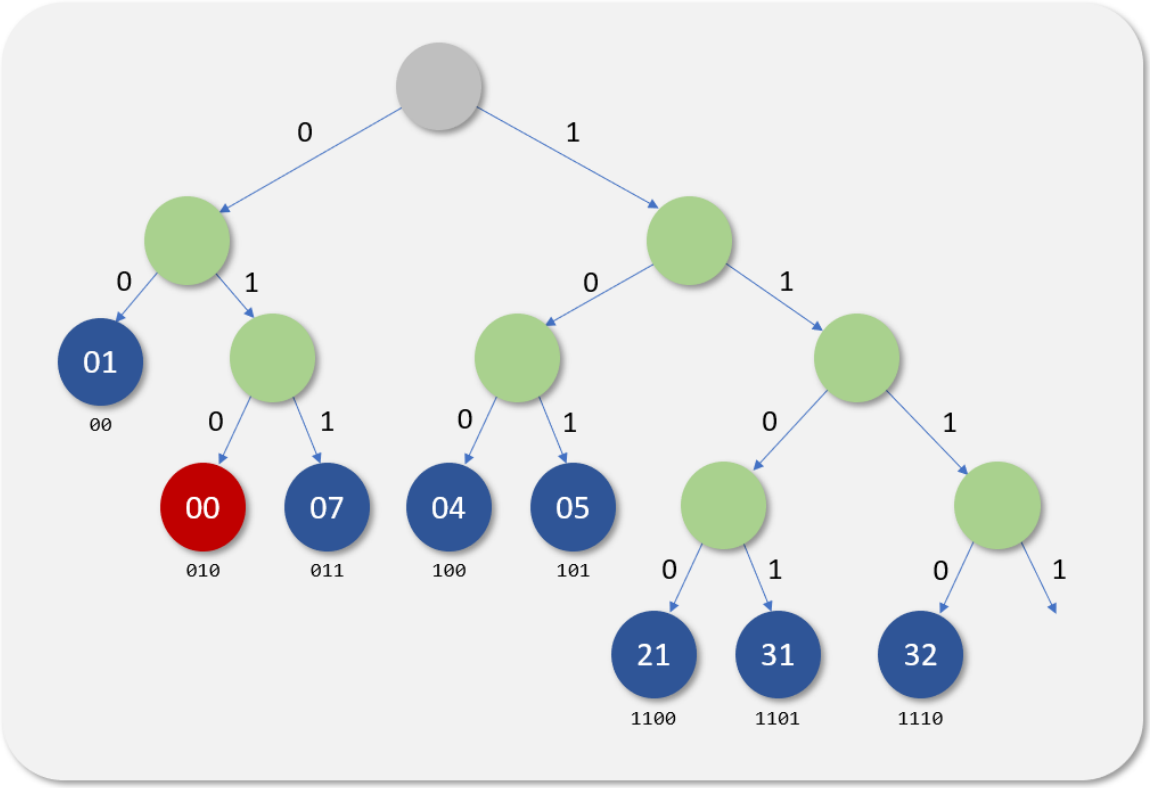
If we encoded the amplitudes using the transform above, without any further information, it would be impossible to delimit the variable length codes and retrieve the data. We must precede these values with more information, in particular the bit width of the following code. Then we know how many bits 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 the table 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 the table is 0, making 16 zeros). These can be chained to represent as many zeros as required. There is also one other special value, `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 a previous part in this document on data compression (Part 1) . A real example of a Huffman tree for encoded values is shown in the figure below:



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 the Huffman tree figure above, 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.

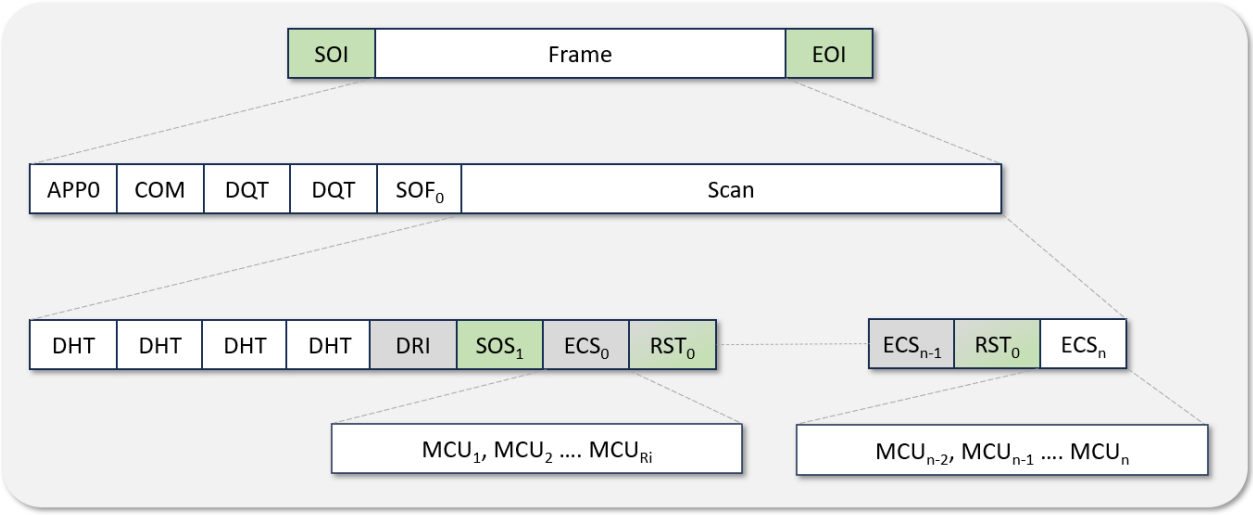
With this step we have finished encoding the data. There is a variable length Huffman code to represent the run-length/bitwidth information, followed by a variable length code representing the quantised DCT values. The Huffman codes themselves (the tree) is not discernible from the encoded data, and 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, 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 the figure below.




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 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 multiple scans can exist separated by a "define-number-of lines" (`DNL`) marker. Again, this is not relevant and not covered in this document.

The actual scan itself consists of a set of optional (according to the JPEG standard) 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 and JFIF 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 precede 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 the above figure 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 is 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 |

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 |

The table 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 |

The above table 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

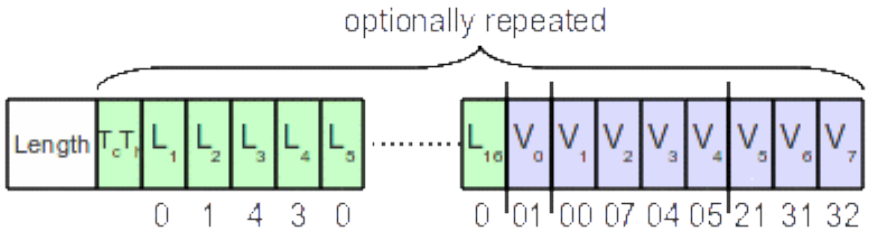
The quantisation tables are simply the 64 divisors used to scale the DCT amplitudes (see above), and the DQT segment simply stores these numbers with a little extra information. The table below shows the format of a DQT segment.

| 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 |

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 the zig-zag figure 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. The table below shows the table format.




In the figure of the Huffman tree in the Huffman Coding section (see above), 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`). The table shows its format.

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

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, as described in Part 1 of this document) or progressive JPEG were avoided. The concepts here are limited to those relevant to the implementation part 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.

# Part 5: JPEG Implementation

## Introduction

In this fifth, and last, part of the document, on data compression, we will look at an implementation, in C++, of a JPEG/JFIF decoder in order to solidify the concepts discussed in the previous part. It is imperative that the previous part has been read and (largely) understood, as the text here assumes this and does not elaborate on the concepts and concentrates on the decoder implementation. It is targeted at those who wish to know, in more detail, the practicalities of implementing JPEG applications and are willing to deep dive into source code. If you've made it this far, well done! So let's begin.

The development of the implementation takes the form of a C/C++ model, but with an architecture that is logic implementation friendly, should that next step be taken in the future, and is based around the DCT algorithm in [5] and pipelined, just as per an intended logic design. A C/C++ model is easier and quicker to implement than an RTL logic implementation and will execute much faster than an RTL simulation. Therefore the model can be tested with a much larger set of test data for a given test period. The model then becomes a specification and test reference for any future logic hardware implementation. As such it is unlikely to compete on performance with software library implementations of JPEG decoders (such as libjpeg-turbo, for example) designed from the start for performance as software running on a computer. The code, along with this documentation, is meant to be instructive though still perform efficiently. A line-by-line description will not be attempted, and this part of the document is meant to be read in conjunction with the open-source source code on github.

## Decoder Software Model Implementation

The software model for the decoder covers all functionality that would also be required in a logic implementation, plus some peripheral code for a command line executable program interface and debugging information etc. As a whole the model will serially process an input JPEG file and output a 24-bit bitmap file. This basic operation is further broken down into processing the header information, and then looping through the 'MCUs', decoding them, and placing the bitmap data in a buffer and outputting the results. The core is based around a class '`jfif`', with a single public method (`jpeg_process_jfif()`). The class header is defined in `jfif_class.h`, and the methods are defined in jfif.cpp. The `jfif` class inherits a

base class, `jfif_idct` that provides the inverse DCT functionality. This is defined in jfif_idct.cpp and `jfif_idct.h`. The basic calling structure of the code is as shown below in the form of pseudo-code. The function names shown match with the actual jfif class methods, and the calling hierarchy is as it is in the model.

```
jpeg_process_jfif()             -- Converts i/p baseline DCT JFIF buffer to 24
                                -- bit bitmap
    jpeg_extract_header()       -- Extracts jpeg header information
                                --   (scan, frame, quant/huffman tables, DRI)
        jpeg_dht()              -- Makes a decode structure from huffman
                                --   table data
    jpeg_bitmap_init()          -- Creates space for appropriately sized
                                -- bitmap and initialises header data

  LOOP for each MCU:            -- jpeg_process_jfif() process an MCU at a
                                --   time until EOI (or error)
    jpeg_huff_decode()          -- Gets an adjusted Huffman/RLE decoded
                                --   amplitude value and ZRLs, or marker or
                                --   end-of-block
        jpeg_dht_lookup()       -- Does huffman lookup on code and extracts
                                --   amplitude data, or flags a marker
            jpeg_get_bits()     -- Gets top bits from barrel shifter and
                                --   (optionally) removes. Pulls in extra
                                --   input if needed.
            jpeg_amp_adjust()   -- Adjusts amplitude to +/- amplitude value

        <dequantise>            -- De-quantisation done in jpeg_huff_decode
                                --   directly from selected table
    jpeg_idct()                 -- Inverse discrete cosine transform

    jpeg_ycc_to_rgb()           -- Converts YCbCr to RGB on an MCU

    jpeg_bitmap_update()        -- Updates bmp data buff with 8x8 RGB values

  ENDLOOP


  return pointer to bitmap
```

The top-level entry point for JFIF decoding is the `jpeg_process_jfif()` method. This has a very simple interface. The first argument is simply a pointer to a buffer of uint8 bytes (`*ibuf`), containing the JFIF data to be decoded. The JFIF data is expected to start in the first byte and continue consecutively for the whole image. No assumptions are made by the `jpeg_process_jfif()` method about the size of the file, or dimensions of the image etc.—all this is calculated from the header.

The second argument is a pointer to a uint8 pointer (`**obuf`). The pointer referenced by the passed in pointer will be updated to point to a buffer containing the decoded bitmap data (assuming no errors). The memory required for the bitmap is dynamically allocated by `jpeg_process_jfif()`, and it is the responsibility of the calling process to free this memory when it is no longer required.

The functions return one of several return codes, defined in `jfif.h`:

- JPEG_NO_ERROR : Decode completed without error
- JPEG_FORMAT_ERROR : A format error was encountered when parsing the header
- JPEG_MEMORY_ERROR : A memory allocation failure occurred
- JPEG_UNSUPPORTED_ERROR : A valid, but unsupported, format was encountered

Within jpeg_process_jfif() the method calls jpeg_extract_header() to parse the JFIF header, and return various pointers to the header information, such as SOS data (scan_header_t *), SOF0 data (frame_header_t *) and reset interval. With this data jpeg_bitmap_init() is called that allocates a buffer for bitmap data (bmp_ptr) and generates a bitmap header for a 24-bit RGB image (even if it's greyscale). It also calculates, from the header data, the number of MCUs it is to process (total_mcus). The main body of the method is then a while loop that processes MCUs one at a time (and markers) until and EOI marker is encountered. The main body of the method jpeg_huff_decode() is a while loop, that loops, calling jpeg_huff_decode() to return either a marker or an MCU consisting of an 8 by 8 array of adjusted amplitude values (scan_data_ptr). The only markers expected to be returned during scan data processing are RSTn and EOI, otherwise an error is returned. Some checks are done of the sequence of RSTn markers received, though the actual resetting of DC values is handled by lower-level functions. The scan_data_ptr data is then inverse discrete cosine transformed (in jpeg_idct()) and then, if a colour image, converted to RBG (jpeg_ycc_to_rgb()). The resulting RGB data is then sent to jpeg_bitmap_update() for constructing into the bitmap buffer. When all MCUs are processed, the **optr is updated to point to the bitmap data, and jpeg_process_jfif() returns with a status.

## Header extraction

JFIF header extraction is performed by the jpeg_extract_header() method. This function is based around a single "switch" statement that parses all valid marker types expected before the SOS marker, though it flags an error if any of those markers are from an unsupported file format. The JFIF format mandates some ordering of segments delineated by the markers but, for robustness, the jpeg_extract_header() method only expects the first item to be an SOI marker, and that an SOS marker marks the end of the header information. Any other segment can come in any order, whatever the JFIF specification might say. It is known that not all images follow the JFIF format completely strictly.

All markers fall in to one of two categories: ones that have no following data (e.g. SOI) and those that do (e.g. `APP0`). The latter are always followed by 2 bytes of length, which dictates the total number of following bytes (including the 2-byte length) of the segment. The main markers of interest to the `jpeg_extract_header()` method are the `SOI`, `SOF0`, `SOS`, `DHT`, `DQT`, `APP0` and `APP14`. The `SOI` has no information, but marks the start of the image, and is expected first, otherwise an error is returned. The `SOF0` segment contains data for image dimensions, whether colour or greyscale and a set of values determining sub-sampling (if any), quantisation table selections and component ID—one for each component of a set (see Part 4). A pointer to a frame_header_t structure is set to the input buffer location 2 bytes from the `SOF0` marker (to skip the length field) and returned in the `**fptr` argument. Note that all other start of frame markers (`SOF1` to `SOF15`) are for unsupported formats, and the method returns an error if they are encountered.

The start of scan segment (`SOS` marker) has information for each component, defining a component selector, a DC Huffman table selector, and an AC Huffman table selector. Unlike for `SOF0`, we cannot simply point to the input buffer with a pointer to a structure as the variable length part of the segment (i.e. the component specific fields, which may have 1 set or 3 sets), as this variable length sub-section is followed by another fixed section, whose offset is dependent on the number of component data sets. Therefore some buffer space is allocated and the data extracted into this buffer, and a pointer returned (`*sptr`) of type `scan_header_t`, that points to this extracted data.

The Huffman table segment (DHT) requires a bit more processing, and so a sub-function (`jpeg_dht`) is called to process segments of this type, returning a pointer of type `DHT_offsets_t` which, in turn is returned by `jpeg_extract_header()`. The aim of the `jpeg_dht()` method is to construct a Huffman table with as few bits of information as possible. This information consists of the 16 lengths of the segment (see Figure 7 in Part 4), the offsets within the `Vn` data where the different bit width codes start, and the end codes for the different bit widths. Note that data for bitwidth n is placed at index n-1 in the data arrays, so, for example, bit width 3 data is at index 2.

As discussed in the Part 4, this is all that is needed to decode the Huffman encoded data. If input data bits are less than a certain bit width's end code, but bigger than the next smallest width's end code, then that is the number of bits of the code. Subtracting the end code of the next smallest bit width, doubled to give the start code of the matching bit width, from the input data bits gives the offset into the

entries for that bit width, and hence the decoded value. Taking the example of from the Part 4, we would have a `DHT_offsets_t` structure filled in as shown:

- `Ln[]` (points directly to Ln values in `dht` buffer, overlaid on the input data buffer, `buf[]`)
  - 0, 1, 4, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
- `vmn_offset[n]` (array of pointers to `dht[]` indexes)
  - 0 ⇒ `NULL`
  - 1 ⇒ `dht[0]` : 01
  - 2 ⇒ `dht[1]` : 00, 07, 04, 05
  - 3 ⇒ `dht[5]` : 21, 31, 32
  - others -> NULL
- `row_break_codes[n]`
  - 0 = `0b`
  - 1 = `01b`
  - 2 = `110b`
  - 3 = `1111b`
  - others don't care

If the input data was a Huffman encoded run-length, and presented, say, `..010101011`, then the first bit (using lsb) is bigger than `row_break_codes[0]` (for 1 bit width), so this is not a match. The first two bits are `11b`, which is bigger than `row_break_codes[1]`, but the first three bits are `011b` which is less than `row_break_codes[2]`, meaning we have a 3-bit code. Taking the endcode for 2 bits (`row_break_codes[1] = 01b`) and doubling it gives `010b`, or 2. Subtracting this from the 3 bits of the input is `011b - 010b = 001b`, or 1. The 3-bit `vmn_offset` pointer (`vmn_offset[2]`) points to `dht[1]`, and the value at offset 1 from this has a value `07h`, and hence is the decoded value—i.e. the variable code `011b` of the input has a byte value of `07h`. The `jpeg_dht()` functions returns a pointer to a `DHT_offsets_t` structure with the values filled in for the Huffman table, having allocated some memory space for table. The `Tc` and `Th` values are extracted from the input, and the method loops through each bit width (smallest to largest), filling in the table entries, based on the Ln lengths, and Vn values in the DHT segment.

The quantisation table segments (`DQT`) define the quantisation table values to be used for the different classes (DC or AC) of data, and IDs (which component). Following the 2-byte length are one or more 65-byte value sets. The first byte consists of two nibbles specifying the `Pq` (always 8) and `Tq` (destination) value. Then follows the 64 bytes for the 8×8 quantisation table values. These are extracted into the appropriate table (indexed by the `Ptq` value) pointed to by `*qptr`, of type `DQT_t`.

The values are turned into integer values at this point as they are pre-scaled with values determined by the Arai et. al methods (see [3], [5]), allowing the saving of a multiplication at that point.

The application sections are really only processed to make some compatibility checks. A JFIF file should have an `APP0` segment, and it ought to be the first segment after `SOI` (but might not be). The first `APP0` should begin with the string "`JFIF\0`", and this is checked. Subsequent `APP0` sections, if any, should begin with the string "`JFXX\0`", and this is also checked. If these criteria are met, then the image is flagged as JFIF (`is_JFIF`). Some JPEG files that are not JFIF might have an `APP14` section, usually associated with Adobe files, and beginning with the string "`Adobe`". If this is the case, and not flagged as JFIF, then colour space information is plucked from a field in the segment and checked whether it is RGB (if 0) or YCbCr (if 1) and flagged as such (`*is_RGB` is set `TRUE` or `FALSE`, accordingly).

All other markers or forms of segments not covered here are treated as errors, and the `jpeg_extract_header()` returns the appropriate error code. All being well, the method returns a good status and the header information in the buffers.

## Huffman Decode

The purpose of the `jpeg_huff_decode()` method is to return either a marker or a set of luminance and, if a colour image, chrominance MCUs that make up a sub-sampled set—i.e. a YCC set of 3 when no sub-sampling, up to a YYYYCC when 4:2:0 sub-sampling. The method makes use of a sub-function `jpeg_dht_lookup()` which returns a structure of type `rle_amplitude_t`, as defined below.

```
typedef struct {
    int  marker;         // if non-zero, hit a marker
    bool is_EOB;         // Flag if EOB
    bool is_ZRL;         // Flag if ZRL
    int  ZRL;            // run length of zeros
    int  amplitude;      // adjusted amplitude (if not EOB or ZRL != 16)
} rle_amplitude_t, *rle_amplitude_pt;
```

The first field (marker) when non-zero contains a valid marker value, indicating that a marker was encountered. The second is a flag indicating that an `EOB` marker terminated the MCU data early. The third give a zeros run-length that occurred before the associated amplitude value (which could be 0). The actual amplitude code, when not EOB or ZRL encountered, is returned in the final field, amplitude.

The jpeg_huff_decode() first does some calculations on the header data to determine the expected number of Y and C MCUs (y_arrays, total_arrays), as well as some sub-sampling factors for use later on. The method then loops for as many MCUs as calculated (indexed by array). Within this outer loop, table selectors are calculated based on whether then MCU is for chrominance or luminance, and the ID etc. The MCU buffers (mcu[][]) are cleared to 0 before commencing to allow for zero run-length skipping and early termination on EOB. The DC value is fetched by calling jpeg_dht_lookup(). This might return a marker, in which case it is processed, and jpeg_huff_decode() returns. When a marker encountered at this point, the jpeg_huff_decode() method returns NULL, and the marker information is returned in the *marker argument. Only one type of marker is expected to be processed in jpeg_huff_decode()—RSTx (others are handled in lower functions or passed up to the calling method). This is for resetting the DC values (which are a running difference), and the DC counts (current_dc_value[]) are cleared on reception of this marker. If not a marker, the appropriate DC value is updated by adding the returned amplitude value to a running total, and de-quantising by multiplying by qptr[Tq][0]. The result is placed in the current mcu[] buffer. After the DC value, the 63 remaining AC values are similarly retrieved, in an inner loop, via jpeg_dht_lookup(), dequantised and put into the mcu[] buffer. If the AC values have a ZRL run-length greater than 0, then the MCU index (mdx) is incremented by that run-length first, before updating the buffer with the de-quantised amplitude. During AC element processing, markers may still be returned. An EOB flagged will simply set mdx to 64 so that the next loop iteration falls through, and effectively skips to the end of that MCU's processing. The only other marker valid to appear during AC data is EOI. This acts like an EOB, but the marker is passed up to the calling method (in *marker). Any other marker is an error. Assuming no markers, when the MCU is complete (with mdx hitting 64), the outer loop iterates to process the next MCU until all MCUs in the group have finished. The jpeg_huff_decode() then returns a pointer to the mcu[] buffer, containing all the mcu data for the whole MCU set.

### Huffman lookup

The jpeg_dht_lookup() method referred to in the last section is the procedure that performs the Huffman lookup as described previously and extracts the following encoded amplitude. As this method is dealing with variable length codes, it works on a barrel shifter variable (jfif_barrel—defined in jfif class() and passed into jpeg_dht_lookup()). On calling the method, matches are made on the barrel shifter bits from their smallest bit width upwards until either a match is made, no match is found (an error) or a marker is found. The code is fetched via the jpeg_get_bits()

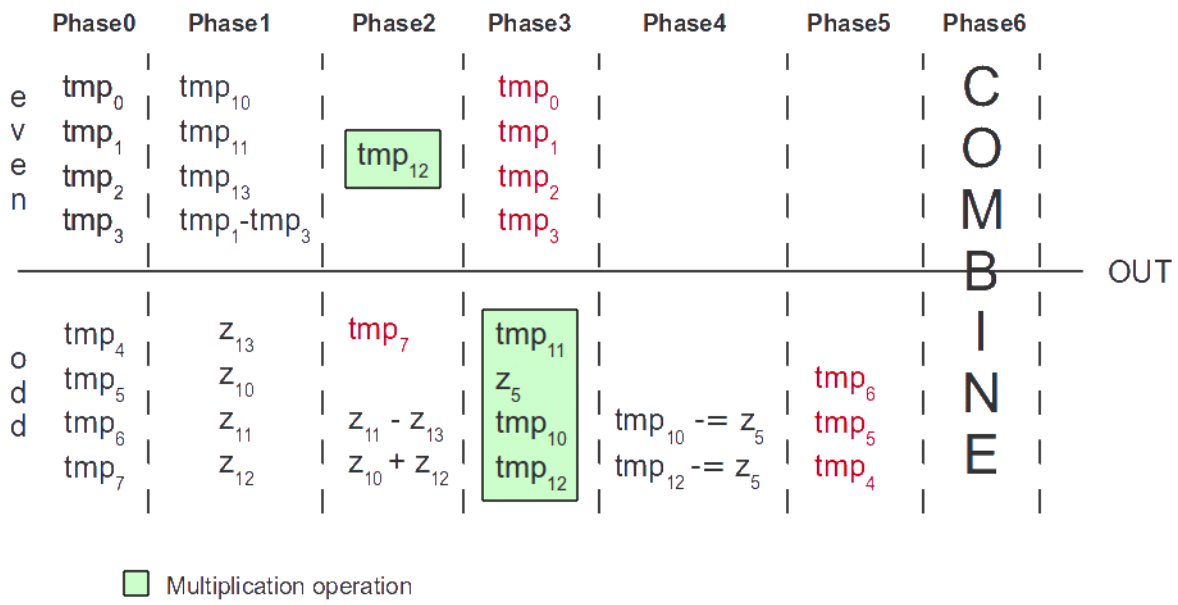
method, which returns the number of bits requested from the barrel. If not enough bits available on the barrel for the requested size it will fetch more bits from the input and add them to the barrel. If it detects a marker (`ffh` followed by a value byte), it will return the marker value ORed with `ffff0000h` to distinguish it from a code. Otherwise the 'n' top bits of the barrel are returned. The bits are removed from the barrel if the '`remove_bits`' argument is set. Until a code match is found, the bits mustn't be removed from the barrel. When the bit width of the code is determined only then is it safe to update the barrel, and delete the bits returned.

Once the `jpeg_dht_lookup()` method has found a matching bit width, it extracts the code (and flags to delete from the barrel) and does the table lookup to get the byte value. The value is checked for being `EOB` or `ZRL`, and the return structure (`rle_amplitude_t`) flags updated if so. If not `EOB` or `ZRL`, the amplitude bitwidth is extracted from the decoded value, and these amplitude bits extracted from the barrel (via `jpeg_get_bits()`) and passed onto `jpeg_amp_adjust()`, which decodes the value to undo the encoding as per Table 2 in Part 4 of the document. Any zero run-length value from the Huffman decoded value is written to the return structure, and this returned to the calling method.

## Inverse DCT

For the inverse DCT, a fast implementation after Arai et. al, and detailed in [3] and [5], has been developed and tested, and the discussions in this part of the document assume this method. If `JPEG_FAST_INT_IDCT` is defined at compile time, this method is selected. In addition, another iDCT method is supplied (`jpeg_idct_slow()`) which is a simple implementation that follows the equations discussed in the previous part and is adapted from [4]. For the slow method, a pair of double loops are implemented for a one-dimensional iDCT of rows and then columns. This method is used if `JPEG_FAST_INT_IDCT` is not defined at compile time. The method can be full precision or be integer fixed point (when compiled with `JPEG_DCT_INTEGER` defined).

The fast method used ([5]) for a one-dimensional transform is pipelined into seven phases and takes the form of that shown in the diagram below:




The actual operations for the phases are not shown in the diagram, but the source code comments reference the variables shown above, for cross-referencing the phase steps, and the operations for each calculation is given alongside the code that implements them.

## Colour Space Conversion and bitmaps

The C/C++ reference model is capable of generating a bitmap output from the JPEG input, converting the YCbCr data to RGB (if not RGB already), and constructing a valid 24bit RGB bitmap. This functionality is not part of the HDL hardware core implementation, but is added to allow test, comparison, and validation in a simple raw standard format. As such, only a fragmentary description will be given here.

The YCbCr conversion to RGB is performed by the `jpeg_ycc_to_rgb()` method. This takes a full set of Y and C MCUs (potentially sub-sampled) and expands into an N-by-N block of RGB triplets—which could be 8×8 if no sub sampling, or 16×8 for 4:2:2 etc. It is passed the final image dimensions and clips the MCU padded data for non MCU aligned boundaries. At the heart of the method is the conversion functions as detailed in Part 4 of the document. The calculations are slightly different for full precision versus fixed point integer (if `JPEG_DCT_INTEGER` defined) but are essentially the same operation. The `jpeg_ycc_to_rgb()` is passed a buffer pointer (`*optr`) to a block of memory (a 2-dimensional array, big enough for a maximally sub-sampled YCC set), in which to place the RGB data.

The `jpeg_ycc_to_rgb()` method only works on the actual image data. Two other functions actually construct the full bitmap file; `jpeg_bitmap_init()` and `jpeg_bitmap_update()`. The first is called after the header extractor routine once

the image dimensions are known. The X and Y dimensions are passed to the method, which allocates memory big enough for the image, including a header, configures the header appropriately, and returns a pointer to the buffer created. `jpeg_bitmap_update()` is called immediately after `jpeg_ycc_to_rgb()`, and takes the buffer returned by that method, and places the data in the appropriate locations of the bitmap buffer, reversing the image from top to bottom to be bottom to top, as for bitmaps. It is this bitmap buffer that is returned by the top-level JPEG method jpeg_process_jfif().

The jpeg_process_jfif() method (and its sub-functions described above) is meant to be used as a stand-alone library class, and also has a C linkage function available, `jpeg_process_jfif_c()`, and can be compiled as such with the provided make scripts. However, a brief mention of the file `jfif_main.c` ought to be made, which uses `jpeg_process_jfif_c()` to produce a command line-based executable for exercising the function for purposes such as verification, generation of bit comparison vectors, etc. Details of the usage of the executable, along with source code download instructions and links, can be found here. The `main()` function in `jfif_main.c` provides code for a simple command line interface, file input and output and also a graphical display window for viewing the images (using the GTK+2 library—not available when compiled with MSVC).

# Conclusions

We have looked at a C/C++ model of a decoder based on the concepts discussed in the previous part. This has solidified these abstract concepts into a practical implementation to demonstrate how real-world implementations are achieved and provide a case study of such implementations. The source code for the implementation can be found on github. The `README.txt` in the repository has details of how to install, compile and run the code, though there are both Windows and Linux compiled binaries available in the repository.

This, then, completes the final part of the document reviewing foundational data compression concepts, both lossless and lossy.

## References

[1] JPEG Standard (JPEG ISO/IEC 10918-1 ITU-T Recommendation T.81)
[2] JPEG File Interchange Format Specification v1.02
[3] Pennebake, W.B. and Mitchell, J.L., 1992, *JPEG Still Image Data Compression Standard*, 1st Ed., ISBN 0-442-01272-1

[4] Nelson, Mark; Gailly, Jean-loup, 1995, *The Data Compression Book*, 2nd Ed., ISBN 1-558-51434-1
[5] Arai, Y., Agui, T., and Nakajima, M.: *A fast DCT-SQ scheme for images*, Trans. IEICE, 1988, E71, (11), pp. 1095–1097