The Quick Chart (.QCT) File Format Specification

Revision 1.03
13 FEB 2011

Craig Shelley
craig@microtron.org.uk

Disclaimer

THIS DOCUMENT AND MODIFIED VERSIONS THEREOF ARE PROVIDED UNDER THE TERMS OF THE GNU FREE DOCUMENTATION LICENCE WITH THE FURTHER UNDERSTANDING THAT:

1. THE DOCUMENT IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE DOCUMENT OR MODIFIED VERSION OF THE DOCUMENT IS FREE OF DEFECTS MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY, ACCURACY, AND PERFORMANCE OF THE DOCUMENT OR MODIFIED VERSION OF THE DOCUMENT IS WITH YOU. SHOULD ANY DOCUMENT OR MODIFIED VERSION PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL WRITER, AUTHOR OR ANY CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY DOCUMENT OR MODIFIED VERSION OF THE DOCUMENT IS AUTHORISED HEREUNDER EXCEPT UNDER THIS DISCLAIMER; AND
2. UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER IN TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL THE AUTHOR, INITIAL WRITER, ANY CONTRIBUTOR, OR ANY DISTRIBUTOR OF THE DOCUMENT OR MODIFIED VERSION OF THE DOCUMENT, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER DAMAGES OR LOSSES ARISING OUT OF OR RELATING TO USE OF THE DOCUMENT AND MODIFIED VERSIONS OF THE DOCUMENT, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES.
3. ALL BRANDS AND PRODUCT NAMES MAY BE TRADEMARKS OR REGISTERED TRADEMARKS OF THEIR RESPECTIVE OWNERS.

Copyright © 2011 Craig Shelley, Mark Bryant

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation Licence, Version 1.2 or any later version published by the Free Software Foundation;

Table of Contents

1.0 Change History....3
2.0 References....4
3.0 Acronyms and Abbreviations....4
4.0 Important Note....4
5.0 Introduction....5
  5.1 Key Features....5
  5.2 QC3 Key Features....6
6.0 Quick Chart File Layout....6
  6.1 Data Formats....6
  6.2 Meta Data....7
    6.2.1 Extended Data Structure....8
    6.2.2 Map Outline....8
    6.2.3 Datum Shift....8
    6.2.4 License Information....8
    6.2.5 Digital Map Shop....9
    6.2.6 Serial Number....9
  6.3 Geographical Referencing Coefficients....10
  6.4 Palette....10
    6.4.1 Interpolation Matrix....11
  6.5 Image Index....11
7.0 Compressed Image Data....12
  7.1 Interlacing....12
  7.2 Pixel Packing....13
  7.3 Run Length Coding....13
  7.4 Huffman Coding....14
    7.4.1 Huffman Codebook....14
    7.4.2 Huffman Bit Stream....15
    7.4.3 Finding the Start of the Bit Stream....15
    7.4.4 Blank Tiles....15
    7.4.5 Huffman Coding Example....16
8.0 Geographical Referencing Polynomials....18
9.0 QC3 Format....19
  9.1 Quick Chart 3 File Layout....19
  9.2 Data Formats....19
  9.3 Meta Data....19
  9.4 Image Index....20
  9.5 Tile Meta Data....20
  9.6 Scale Levels....20
  9.7 Checksum....21
  9.8 Compression....22
    9.8.1 Huffman Codebook....22
    9.8.2 Run Length Encoding....22
    9.8.3 Decompression Example....23
  9.9 Digital Rights Management....24

1.0 Change History

Revision	Date	Author	Description of Change
1.00	01 NOV 2008	Craig Shelley	Initial Issue
1.01	07 MAR 2009	Craig Shelley	Corrected mistakes in the geographical referencing coefficients and equations. This affects the following sections: Section 6.3: Coefficients in columns 3 and 4 swapped. Extra coefficients added for 3rd order polynomials. Section 8.0: Calculation method corrected. Added method for reverse calculation.
1.02	12 JUL 2009	Craig Shelley	Section 7.4.4: Renamed section to Blank Tiles, and amended section body to describe blank tile decoding procedure. Section 7.4.5: Corrected mistakes in Huffman example bit stream data.
1.03	13 FEB 2011	Mark Bryant	Section 2.0: Added new references. Section 3.0: Added acronyms. Section 6.2: Added new meta data fields. Section 9.0: Added QC3 format documentation.

2.0 References

1 http://en.wikipedia.org/wiki/Run-length_encoding

2 http://en.wikipedia.org/wiki/Huffman_coding

3 http://en.wikipedia.org/wiki/WGS-84

4 http://en.wikipedia.org/wiki/Endianness

5 http://en.wikipedia.org/wiki/IEEE-754

6 http://en.wikipedia.org/wiki/Ascii

7 http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

8 http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

3.0 Acronyms and Abbreviations

Acronym/Abbreviation	**Full Meaning**
QCT	Quick Chart
HEX	Hexadecimal
PDA	Personal Digital Assistant
GPS	Global Positioning System
WGS	World Geodetic System
LSB	Least Significant Bit
MSB	Most Significant Bit
ASCII	American Standard Code for Information Interchange
IEEE	Institution of Electrical and Electronics Engineers
QC3	Quick Chart 3 File
RAM	Random Access Memory
AES	Advanced Encryption Standard
ECB	Electronic Codebook Mode
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator

4.0 Important Note

This document has been created with the intention to document a previously undocumented file format. The information contained within has been obtained by painstakingly viewing and attempting to interpret the content of freely available Quick Chart files in a HEX editor, and is therefore is based entirely upon assumptions and guesswork. It is highly likely that mistakes have been made during the compilation of this document.

The information contained within this document is incomplete.

The information contained within this document was NOT obtained by means of reverse engineering software applications.

5.0 Introduction

The Quick Chart file format (.QCT file extension) is a raster image format designed for storing high resolution maps. Its key design goal is to allow the images to be viewed with very little processing overhead, while having an appreciable level of lossless compression. Geographical data embedded within the header allows the image to be mapped in terms of latitude and longitude.
This makes the Quick Chart file format ideally suited for displaying maps on portable devices such as PDAs and hand held GPS units.

5.1 Key Features

- 128 colour palette
- Tile based compression
- Optimum compression algorithm used for each tile
  - Pixel Packing
  - Run Length Encoding [2]
  - Huffman Coding [2]
- Tile index improves performance while displaying a small image area
- Tile Interlacing improves rendering performance while scaling down
- Interpolation Matrix defines how colours should be interpolated
  - Improves rendering performance while scaling down
  - Improves image clarity by allowing certain colours to take precedence over others, for example major roads.
- File header fields define many aspects of the image data including;
  - Title, Name, Ident
  - Revision, Keywords, Copyright
  - Scale, Datum, Projection
  - Depths, Heights
  - Original File Name, Original File Size, Creation Time
  - Map Outline
- Geographical data fields provide polynomial coefficients to allow the conversion between cartesian image co-ordinates and WGS-84 [2] latitude and longitude.
- File format ideally suited for Memory Mapped I/O

A new version of the format (referred to as QC3) stores the image data in a separate file with extension .qc3 whilst retaining the same format for the meta data.

5.2 QC3 Key Features

- Support for files over 4GiB allowing seamless maps of very large areas
- Improved compression algorithm utilising Huffman Coding and Run Length Encoding
- Digital Rights Management
- Large map support for devices with limited RAM
- Improved support for scaled down display

6.0 Quick Chart File Layout

Offset	Size (Bytes)	Content
0x0000	24×4	Meta Data - 24 Integers/Pointers
0x0060	40×8	Geographical Referencing Coefficients - 40 Doubles
0x01A0	256×4	Palette - 128 of 256 Colours
0x05A0	128×128	Interpolation matrix
0x45A0	$w\times h\times4$	Image Index Pointers - QC3 files omit this
-	-	File Body - Text strings and compressed image data

6.1 Data Formats

Integers are stored as 4 bytes in Little-Endian [2] byte order.

Pointers are stored as 4 bytes in Little-Endian byte order and refer to byte locations relative to the beginning of the file.

Doubles are stored as 8 byte IEEE-754 [2] double precision format.

Strings are stored as 1 byte per character in ASCII [2] and are NULL terminated.

Palette colours are stored as 3 bytes + 1 padding byte (Blue, Green, Red, 0x00).

6.2 Meta Data

Offset	Data Type	Content
0x00	Integer	Magic Number 0x1423D5FE - Quick Chart Information 0x1423D5FF - Quick Chart Map
0x04	Integer	File Format Version 0x00000002 – Quick Chart 0x00000004 – Quick Chart supporting License Management 0x20000001 – QC3 Format
0x08	Integer	Width (Tiles)
0x0C	Integer	Height (Tiles)
0x10	Pointer to String	Long Title
0x14	Pointer to String	Name
0x18	Pointer to String	Identifier
0x1C	Pointer to String	Edition
0x20	Pointer to String	Revision
0x24	Pointer to String	Keywords
0x28	Pointer to String	Copyright
0x2C	Pointer to String	Scale
0x30	Pointer to String	Datum
0x34	Pointer to String	Depths
0x38	Pointer to String	Heights
0x3C	Pointer to String	Projection
0x40	Integer Bit-field	Flags Bit 0 - Must have original file Bit 1 - Allow Calibration
0x44	Pointer to String	Original File Name
0x48	Integer	Original File Size
0x4C	Integer	Original File Creation Time (seconds since epoch)
0x50	Integer	Reserved, set to 0
0x54	Pointer to Struct	Extended Data Structure (see section 6.2.1)
0x58	Integer	Number of Map Outline Points
0x5C	Pointer to Array	Map Outline (see section 6.2.2)

6.2.1 Extended Data Structure

Offset	**Data Type**	**Content**
0x00	Pointer to String	Map Type
0x04	Pointer to Array	Datum Shift (see section 6.2.3)
0x08	Pointer to String	Disk Name
0x0C	Integer	Reserved, set to 0
0x10	Integer	Reserved, set to 0
0x14	Pointer to Struct	License Information (Optional, see section 6.2.4)
0x18	Pointer to String	Associated Data
0x1C	Pointer to Struct	Digital Map Shop (Optional, see section 6.2.5)

6.2.2 Map Outline

The Map Outline is an array of latitude/longitude pairs which mark out the boundary of the mapped region. The size of this array is stored in the Meta Data area at offset 0x58. Most maps will have four points to mark out the four corners of the map. Aerial photographs can have several hundred points to mark out the boundaries of counties.

Offset	**Data Type**	**Content**
0x00	Double	Latitude
0x08	Double	Longitude
...	...	...

6.2.3 Datum Shift

The datum shift is a pair of latitude/longitude values indicating the offset on the co-ordinates for this map.

Offset	**Data Type**	**Content**
0x00	Double	Datum Shift North
0x08	Double	Datum Shift East

6.2.4 License Information

The license information contains a description of the product and when combined with a license allows decryption of encrypted map contents.

Offset	**Data Type**	**Content**
0x00	Integer	Identifier of the license – This correlates with name of the license file used that must be paired with the map
0x04	Integer	Unknown
0x08	Integer	Unknown

0x0C	Pointer to String	License description
0x10	Pointer to Struct	Serial Number (see section 6.2.6)
0x14	Integer	Unknown
0x18	16 Bytes	Unknown, set to 0
0x28	64 Bytes	Unknown

6.2.5 Digital Map Shop

QC3 format maps support streaming tiles over HTTP and this structure contains the necessary information.

Offset	Data Type	Content
0x00	Integer	Structure size, set to 8
0x04	Pointer to String	Partial URL to QC3 map file

6.2.6 Serial Number

Offset	Data Type	Content
0x00	32 Bytes	Unknown

6.3 Geographical Referencing Coefficients

The geographical referencing coefficients are a set of polynomial coefficients that enable the map pixel coordinates to be converted into WGS-84 latitude and longitude. All coefficients are stored in double precision format.

Offset	0x00	0x50	0xA0	0xF0
0x00	eas	nor	lat	lon
0x08	easY	norY	latX	lonX
0x10	easX	norX	latY	lonY
0x18	easYY	norYY	latXX	lonXX
0x20	easXY	norXY	latXY	lonXY
0x28	easXX	norXX	latYY	lonYY
0x30	easYYY	norYYY	latXXX	lonXXX
0x38	easYYX	norYYX	latXXY	lonXXY
0x40	easYXX	norYXX	latXYY	lonXYY
0x48	easXXX	norXXX	latYYY	lonYYY

Refer to section 8 for the set of geographical referencing polynomials.

6.4 Palette

The colour palette contains all colours used within the image, allowing colours within the compressed image data to be referenced by colour index. The colour palette can contain up to 256 colours however the image compression algorithms only allow the use of 128 colours. Therefore only the first 128 colours of the palette contain actual colour values. Colours with an index of 129 to 255 are unused and are set to 0.

Each colour in the palette occupies 4 bytes, therefore the total palette size including the 128 unused colours is 1024 bytes.

Offset	**Data Type**	**Content**
0x00	Byte	Blue Intensity [0-255]
0x01	Byte	Green Intensity [0-255]
0x02	Byte	Red Intensity [0-255]
0x03	Byte	Padding byte, set to 0

6.4.1 Interpolation Matrix

Since the images stored in Quick Chart format are relatively large, it is often necessary to view them scaled down. To improve the clarity of the scaled down image, interpolation is required. However due to the size of the image, and the fact that a palette is used, this process would require a high processing overhead.

The solution to this problem is to use a matrix of pre-calculated colour indices. To determine the palette index of two interpolated colours, use the two colours as a row and column index of the interpolation matrix.

The image in *figure 1* represents an example content of an interpolation matrix. An extra row of pixels has been added to the top and left to show the original colours. The matrix has a line of symmetry about the leading diagonal. Also notice how the colours of the matrix are biased away from background colours towards the colours of major map features such as grid lines (blue), roads (blue, pink, orange and yellow) and text (black). This biasing allows these features to remain visible as the image is scaled down, preserving the clarity of the map.

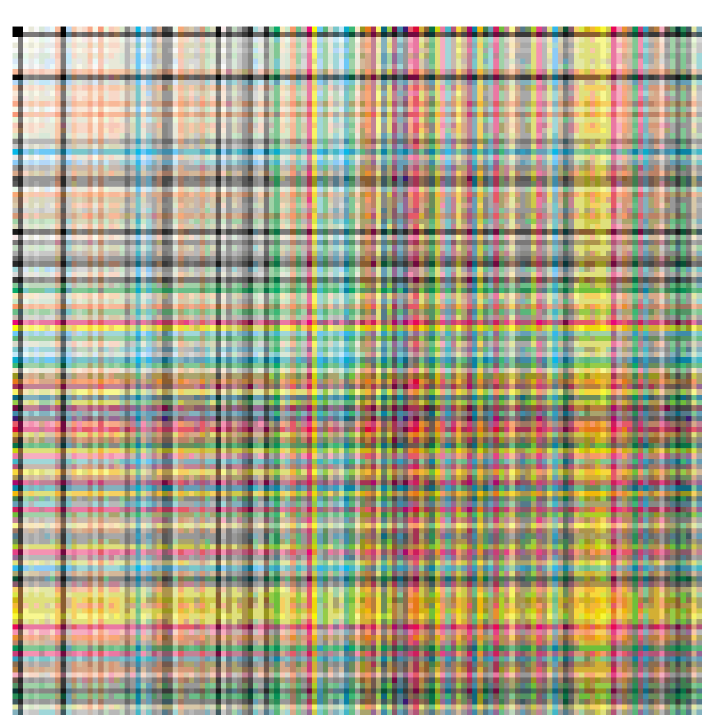

Figure 1: Interpolation Matrix

Interpolation matrix data are stored as a block of 16384 bytes, one byte per colour as 128 rows of 128 colours. The offset of any given row/column index into this matrix can be determined by;

$$\text{Offset} = (128 \times y) + x$$

Symmetry of the matrix allows interchangeability of the x and y variables.

6.5 Image Index

The image is compressed using tiles of 64x64 pixels. The image dimensions in terms of the number of tiles are stored in the Height and Width Meta data fields. For each tile in the image, an entry is present in the index. Each index entry is a pointer to the compressed image data for the tile. Each pointer is four bytes in size, hence the size of the image index can be calculated as follows;

$$\text{Index Size} = 4 \times \text{width} \times \text{height}$$

The offset of any given tile in the index can be calculated by;

$$\text{Offset} = 4 \times ((\text{Width} \times y) + x)$$

7.0 Compressed Image Data

Each tile within the image is compressed to reduce the size of the image file. Three compression algorithms are used to compress the image data, and all three algorithms rely on the fact that an image tile will likely contain fewer colours than the overall image. By using a sub-palette, the number of bits required to reference a given colour can be reduced.

- Pixel Packing

  Not strictly a compression technique, a more efficient method for storing tiles that have few colours.

- Run Length Coding

  More effective on tiles where a few colours are repeated many times.

- Huffman Coding

  An entropy encoding algorithm, useful for complex tiles where some colours occur much more frequently than others.

The pixel packing and run length coding algorithms use the first byte of the compressed data to determine the number of colours in the sub-palette. The encoding of this first byte of compressed data enables it to also be used to determine the compression algorithm used by a tile, as follows;

If $b_0 == 0$ or $b_0 == 255$ then tile is Huffman coded.

Else If $b_0 > 127$ then tile is Pixel Packed.

Else tile is Run Length coded.

7.1 Interlacing

The pixel content of each tile is scanned from left to right in rows of 64 pixels. The rows however are not in top to bottom order. Instead they are interlaced using a bit-reverse sequence. The decompressed tile data must be scanned out row at a time using this row sequence;

Original Row	Binary	Reverse Binary	Destination Row
0	000000	000000	0
1	000001	100000	32
2	000010	010000	16
3	000011	110000	48
4	000100	001000	8
5	000101	101000	40
...	...	...	...

This interlacing scheme reduces the overheads while viewing an image at scaled down resolutions. For example, if an image was to be displayed at a scale down of 1:2, it would be desirable to take every other row of each tile. With this interlacing scheme, to display all of the even rows, only the first half of the tile data needs to be decompressed. This works for all scale down factors, e.g. for 1:4, only the first ¼ of the tile data needs to be decompressed.

7.2 Pixel Packing

Pixel packing is an efficient method for storing tile data where the number of colours used in the tile is low. A sub-palette is created for the colours used within the tile, and the number of bits required per pixel is determined from the size of the sub-palette. For example, if a tile had 13 colours, 4 bits would be required in order to address the colours of the sub-palette.

256 - Subpal Size	Colour	Colour	Colour	Colour	Colour	...	Packed Tile Data	Packed Tile Data	Packed Tile Data	Packed Tile Data	Packed Tile Data	Packed Tile Data	Packed Tile Data	Packed Tile Data	...	...

The first byte codes for the size of the sub-palette;

$$\text{Sub-Palette Size} = 256 - b_0$$

Following this is the sub-palette with one byte per colour index. Each sub-palette byte is a colour index for the main image palette. The tile image data immediately follows the sub-palette.

Tile image data are stored in blocks of 4 bytes. The colours are then packed into the block of 4 bytes, aligned to the LSB of the first byte. Any remaining space within the 4th byte that is too small to fit another pixel is not used. For example, if a tile has a sub-palette of 7 colours, 3 bits would be required per pixel. Therefore 10 pixels (30 bits) of data can be packed into the 4 byte block, leaving 2 unused bits. In this case, a total of 410 blocks (1640 bytes) of data would be required in order to store the 64x64 pixel tile.

7	Byte 0						0	7	Byte 1						0	7	Byte 2						0	7	Byte 3						0
3	3	2	2	2	1	1	1	6	5	5	5	4	4	4	3	8	8	8	7	7	7	6	6	X	X	10	10	10	9	9	9

The number of bits required to store the colour is determined from the sub-palette size.

7.3 Run Length Coding

Run length coding compresses image data where a single colour is repeated multiple times. By using a sub-palette similar to the pixel packing technique, the number of bits required to reference a colour is reduced, leaving the remaining bits of the byte to be used for specifying the repeat count.

Subpal Size	Colour	Colour	Colour	Colour	Colour	...	Repeat Count / Colour	Repeat Count / Colour	Repeat Count / Colour	Repeat Count / Colour	Repeat Count / Colour	Repeat Count / Colour	Repeat Count / Colour	Repeat Count / Colour	...	...

The first byte codes for the size of the sub-palette. However with run length coding, the number does not need to be subtracted from 256.

Following this is the sub-palette with one byte per colour index. Each sub-palette byte is a colour index for the main image palette. The tile image data immediately follows the sub-palette.

Compressed data are stored as a stream of bytes, with the sub-palette index right aligned to the LSB and the repeat count stored in the remaining bits. For example, if a tile only used 4 colours, the 4 colour sub-palette would only require a 2 bits in order to address each colour. This leaves 6 bits to be used to specify the repeat count, giving maximum of 63 repeats.

7	Run-length Encoded Byte						0
REP5	REP4	REP3	REP2	REP1	REP0	COL1	COL0

The number of bits required to store the colour depends only upon the sub-palette size.

7.4 Huffman Coding

Huffman coding is the most complex of the three coding methods. A special sub-palette called a code book is used to reduce the number of bits per pixel, however the binary codes used to reference the colours of the sub-palette have varying lengths. Short code words are given to colours that occur most frequently, and the longest code words are given to colours that occur least frequently.

During the compression process, the tile is analysed to determine the frequency of occurrence of each colour. A Huffman tree is then created by sorting the frequencies in order of occurrence, and leafs and branches are assigned systematically such that the most frequently occurring colours have the shortest route from the root node to the leaf. The method produces an optimised Huffman tree which can then be transformed into a codebook with varying code lengths for compressing the tile image data.

To decompress the tile, the same codebook created for the compression process must used. Image data are treated as a continuous bit stream and as each bit is read, the codebook is used to determine which direction to take at a certain branch of the Huffman tree. Once a leaf node is reached, the colour for the current pixel has been determined, codebook pointer is returned to the root node of the tree and execution of the bit stream sequence continues.

0x00 or 0xFF	Code Book Data	Code Book Data	Code Book Data	Code Book Data	Code Book Data	Code Book Data	...	Bit Stream Data	Bit Stream Data	Bit Stream Data	Bit Stream Data	Bit Stream Data	Bit Stream Data	Bit Stream Data	...	...

7.4.1 Huffman Codebook

The codebook is stored in such a way that it resembles the Huffman tree. Each entry in the codebook is either a branch or a colour. A simple test can determine if a codebook entry b_n is a branch or a colour;

If $b_n < 128$ then b_n is a colour index from the main palette.

Else b_n is a branch.

Codebook branches are essentially a relative 'goto' instruction. The branch entry contains the destination address relative to the current position in the code book. Codebook branches always jump forwards in the codebook and one of take two forms, *near* or *far*. With *near* branches, the relative jump destination is within 127 bytes of the jump origin. *Far* branches can have jump destinations up to 65536 bytes away from the jump origin. In reality, it is unusual to have a far jump much above 200 for a tile of 64x64 pixels. If a codebook entry b_n is a branch, then the type and relative jump distance of the branch can be calculated as follows;

If $b_n > 128$ then the branch is of type *near*.

$$\text{Relative Jump} = 257 - b_n$$

If $b_n == 128$ then the branch is of type *far*.

$$\text{Relative Jump} = 65537 - (256 \times b_{n+2} + b_{n+1}) + 2$$

Note that the *far* jump requires an additional 2 bytes of space in the codebook.

7.4.2 Huffman Bit Stream

Image data are packed into a continuous binary stream beginning in the LSB of each byte. As each bit is read from the bit stream, the Huffman tree within the codebook section is traversed in a similar manner as the execution of a finite state machine. At each branch in the tree, the bit read from the bit stream determines whether or not to follow the branch, or continue;

- 0 - Do not follow the branch, continue to the next entry.
- 1 - Follow the branch.

As soon as a colour is encountered in the codebook, the colour is output for the current pixel, and the codebook pointer must be reset to the beginning of the codebook.

The process should be stopped after the last pixel of the tile has been drawn.

7.4.3 Finding the Start of the Bit Stream

In order to find the start of the bit stream, a codebook tree scan must be performed.

Since every branch in the tree splits two ways, there will always be one more colour than the total number of branches. By scanning through the codebook, counting the number of branches, and the number of colours. The end of the codebook is at the point where the number of colours found exceeds the number of branches found.

The end of the codebook is the beginning of the bit stream.

7.4.4 Blank Tiles

Blank tiles can be decoded using the normal Huffman decoding procedure. A blank tile has no branches in its codebook, and is therefore a single colour connected to the root node. The tile is a blank square filled with this colour, and the bit stream has zero length.

7.4.5 Huffman Coding Example

To decode the following compressed data;

00	F7	FF	54	FF	34	FF	1D	FF	53	2F	1B	35	F2	AB	06	78	E4	...

First parse through, identifying the sub-palette colours and and branches, then identify the beginning of the bit stream using the technique described in section 7.4.3;

	Code Book Data											Bit Stream Data						
00	F7	FF	54	FF	34	FF	1D	FF	53	2F	1B	35	F2	AB	06	78	E4	...

Example sub-palette mapping;

Palette Index	Example Colour
0x54	Red
0x34	Green
0x1D	Blue
0x53	Magenta
0x2F	Cyan
0x1B	Yellow

The Huffman codebook can then be checked for out of bounds branch jumps by verifying that all jumps fall within the codebook. The codebook could also be checked to verify that only one route leads to any given branch or colour. Refer to *figure 2* for a diagrammatic representation of the codebook.

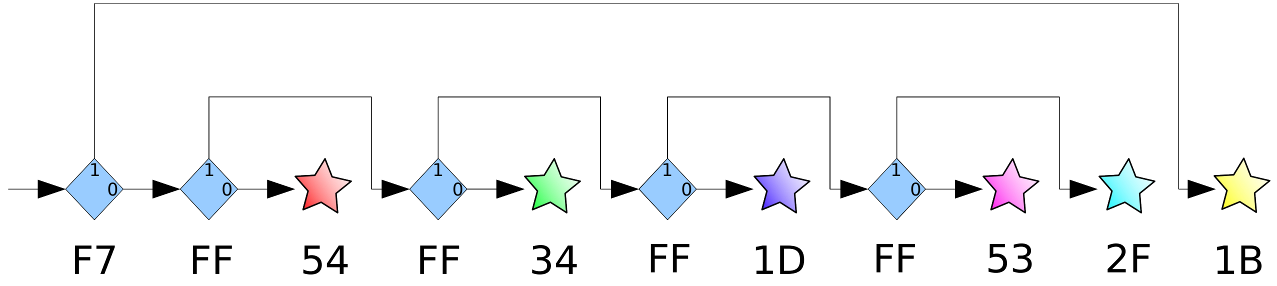

Figure 2: Diagrammatic Representation of Huffman Codebook

From section 7.4.1:

0xF7 decodes as a relative jump forwards of 10 bytes.

0xFF decodes as a relative jump forwards of 2 bytes.

The Huffman tree represented by this codebook can then be drawn, refer to *figure 3*.

From the Huffman tree, the variable length codes that represent each colour can be easily identified.

Palette Index	**Example Colour**	**Huffman Code**
0x54	Red	00
0x34	Green	010
0x1D	Blue	0110
0x53	Magenta	01110
0x2F	Cyan	01111
0x1B	Yellow	1

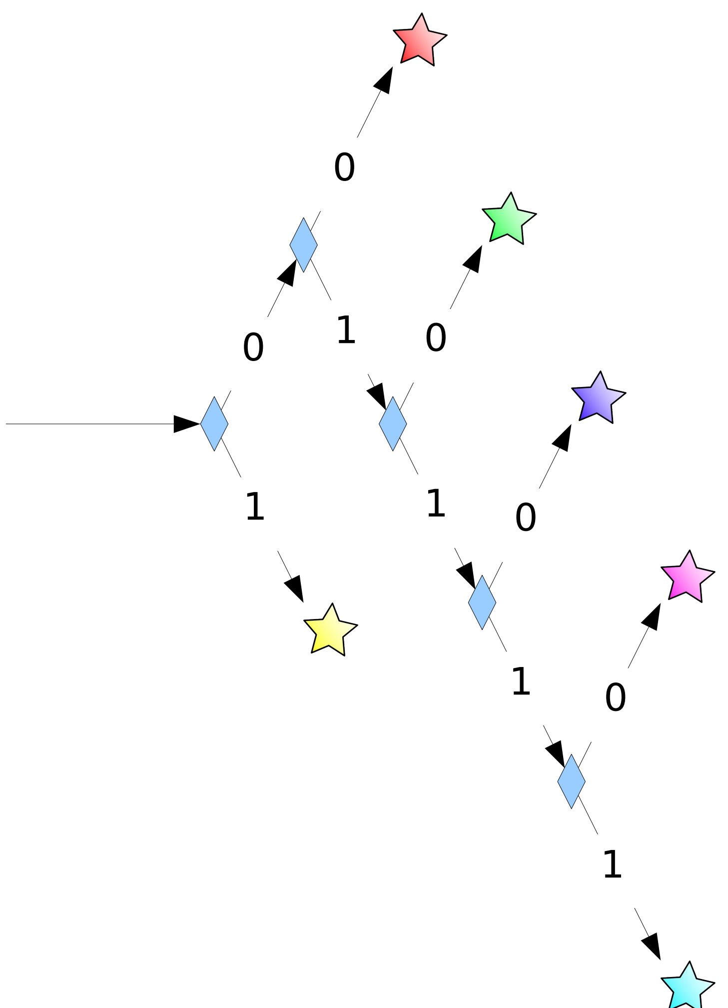

Figure 3: Huffman Tree

The bit stream can then be decompressed by processing one bit at a time until each colour has been decoded. Using the example bit stream data;

Bit Stream Data						
35	F2	AB	06	78	E4	...

Expanding the bit stream (from left to right);

0 0 1 1 0 1 0 1	1 1 1 1 0 0 1 0	1 0 1 0 1 0 1 1	0 0 0 0 0 1 1 0	0 1 1 1 1 0 0 0	1 1 1 0 0 1 0 0

Decoded colours

0 0	1	1	0 1 0	1	1	1	1	1	0 0	1	0 1 0	1	0 1 0	1	1	0 0	0 0	0 1 1 0	0 1 1 1 1	0 0	0 1 1 1 0	0 1 0	0

Therefore 23 pixels have been compressed into 6 bytes.

Due to the method by which the codebook is stored, the decompression algorithm does not need to extract and build the Huffman tree for each tile. After verifying the integrity of the codebook, the branches of the Huffman tree can be followed by evaluating it “in situ” while processing the bit stream, refer to *figure 2*.

8.0 Geographical Referencing Polynomials

The geographical referencing coefficients enable forward and reverse transformation between image pixel coordinates and WGS-84 longitude and latitude. The image coordinates x and y are relative to the top left corner of the image.

The set of polynomial coefficients and their location within the file was discussed in section 6.3.

Converting image x/y coordinates to longitude/latitude:

$$\begin{aligned}\lambda = {} & \mathtt{lonXXX} \times x^3 + \mathtt{lonXX} \times x^2 + \mathtt{lonX} \times x + \\ & \mathtt{lonYYY} \times y^3 + \mathtt{lonYY} \times y^2 + \mathtt{lonY} \times y + \\ & \mathtt{lonXXY} \times x^2 y + \mathtt{lonYYX} \times y^2 x + \mathtt{lonXY} \times xy + \\ & \mathtt{lon}\end{aligned}$$

$$\begin{aligned}\phi = {} & \mathtt{latXXX} \times x^3 + \mathtt{latXX} \times x^2 + \mathtt{latX} \times x + \\ & \mathtt{latYYY} \times y^3 + \mathtt{latYY} \times y^2 + \mathtt{latY} \times y + \\ & \mathtt{latXXY} \times x^2 y + \mathtt{latYYX} \times y^2 x + \mathtt{latXY} \times xy + \\ & \mathtt{lat}\end{aligned}$$

Converting longitude/latitude to image x/y coordinates:

$$\begin{aligned}x = {} & \mathtt{easXXX} \times \lambda^3 + \mathtt{easXX} \times \lambda^2 + \mathtt{easX} \times \lambda + \\ & \mathtt{easYYY} \times \phi^3 + \mathtt{easYY} \times \phi^2 + \mathtt{easY} \times \phi + \\ & \mathtt{easXXY} \times \lambda^2 \phi + \mathtt{easYYX} \times \phi^2 \lambda + \mathtt{easXY} \times \lambda \phi + \\ & \mathtt{eas}\end{aligned}$$

$$\begin{aligned}y = {} & \mathtt{norXXX} \times \lambda^3 + \mathtt{norXX} \times \lambda^2 + \mathtt{norX} \times \lambda + \\ & \mathtt{norYYY} \times \phi^3 + \mathtt{norYY} \times \phi^2 + \mathtt{norY} \times \phi + \\ & \mathtt{norXXY} \times \lambda^2 \phi + \mathtt{norYYX} \times \phi^2 \lambda + \mathtt{norXY} \times \lambda \phi + \\ & \mathtt{nor}\end{aligned}$$

Where:

x	is the pixel horizontal coordinate from the left
y	is the pixel vertical coordinate from the top
λ	is the WGS-84 Longitude
ϕ	is the WGS-84 Latitude

The datum shift east/north values from the meta data must be added to the converted longitude/latitude after converting from x/y coordinates. Similarly, the datum shift must be subtracted from the latitude/longitude before converting back to x/y coordinates.

9.0 QC3 Format

The QC3 format stores the meta data in a file with extension .qct, and the image data in a file with the same name but with extension .qct. The following differences can be found in the Quick Chart File:

- The version number is 0x20000001
- There is no Image Index present

9.1 Quick Chart 3 File Layout

Offset	Size (Bytes)	Content
0x0000	10×4	Meta Data - 24 Integers
0x0028	$w \times h \times 8$	Image Index Pointers
-	-	File Body - compressed image data

9.2 Data Formats

Integers are stored as 4 bytes in Little-Endian byte order.

Pointers are stored as 8 bytes in Little-Endian byte order and refer to byte locations relative to the beginning of the file.

Sizes are stored as 4 bytes in Little-Endian byte order, but need to be multiplied by 4 to obtain the size in bytes.

9.3 Meta Data

Offset	Data Type	Content
0x00	Integer	Magic Number, set to 0x484DF282
0x04	Integer	File Format Version, set to 1
0x08	Integer	Encryption Scale (see section 9.9)
0x0C	Integer	Width (Tiles)
0x10	Integer	Height (Tiles)
0x14	5 Integers	Reserved, set to 0

9.4 Image Index

The image is compressed using tiles of 1024x1024 pixels. The image dimensions in terms of the number of tiles are stored in the Height and Width Meta data fields. For each tile in the image, an entry is present in the index. Each index entry is a pointer to the tile meta data. Each pointer is eight bytes in size, hence the size of the image index can be calculated as follows;

$$\text{Index Size} = 8 \times \text{width} \times \text{height}$$

The offset of any given tile in the index can be calculated by;

$$\text{Offset} = 8 \times ((\text{Width} \times y) + x)$$

There are two reserved values for pointers:

0	means the tile is missing and may be streamed if possible
-1	means the tile is not defined and does not exist

9.5 Tile Meta Data

Each pointer in the image index points to this structure.

Offset	Data Type	Content
0x00	10 Sizes	Size of the compressed image data for each scale starting from the Huffman Tree size (see section 9.6)
0x28	Size	Size of of the compressed image data starting from the Huffman Tree size
0x2C	10 Bytes	One byte checksum for each scale (see section 9.7)
0x36	22 Bytes	Reserved, set to 0
0x4C	Size	Size of the Huffman Tree (see section 9.8)
0x50	...	Huffman Tree (see section 9.8)
...	...	Compressed Image Data (see section 9.8)

9.6 Scale Levels

QC3 files used the same interlacing pattern as described in section 7.1 with the exception that there are now 1024 rows of 1024 pixels instead of 64. The row pattern is thus:

Original Row	Binary	Reverse Binary	Destination Row
0	0000000000	0000000000	0
1	0000000001	1000000000	512
2	0000000010	0100000000	256
3	0000000011	1100000000	384
4	0000000100	0010000000	128
5	0000000101	1010000000	640
...	...	...	...

In order to read the minimum amount of data from disk when scaled down, the meta data contains an array of sizes of the compressed image data (including the Huffman Tree and Huffman Tree size) that needs to be read to display the relevant number of rows.

Offset	**Number of rows the size represents**
0x00	1024
0x04	512
0x08	256
...	
0x24	2

For example, if one wanted to read the first 256 rows (subject to interlacing), they would look up the value of offset 0x08 in the meta data, multiply it by 4, and then read that many bytes starting at offset 0x4C (the size of the meta data excluding to the Huffman Tree size). When decompressing the data, they can be sure that they have a sufficient amount the compressed stream to read the 256 rows in their entirety.

To clarify, the compressed data forms one stream for all 1024 rows (although interlaced). However when scaling down the image, one does not need all data, and the size array describes how much of the compressed data one needs.

9.7 Checksum

The data needed to be read for each scale level also has a one byte checksum. The order is the same as the size array in that the first checksum corresponds to the entire 1024 rows. The calculate the checksum, perform the following algorithm:

- Compute the sum of the first *size* 4 byte integers from offset 0x4C of the tile meta data, where *size* is the value in the size array
- Compute the sum of the 4 bytes in the previous value, and take the least significant byte

For example, calculate the checksum of the following compressed data where the size field we are interested in has the value 3;

00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	11	...

First integer = 0x33221100

Second integer = 0x77665544

Third integer = 0xBBAA9988

Sum (truncated to fit an integer) = 0x6632FFCC

Sum of bytes = 0x66 + 0x32 + 0xFF + 0xCC

Sum = 0x63

9.8 Compression

Tiles are compressed using a combination of Huffman Coding and Run Length Encoding. The first integer following the tile meta data is the size of the Huffman tree. This is used in contrast to Quick Chart Huffman tree, as data must be aligned to multiples of 4 bytes, thus the start of the code book and bit stream fall on 4 byte boundaries.

Size LSB	Size	Size	Size MSB	Code Book Data	Code Book Data	Code Book Data	Code Book Data	Code Book Data	...	Bit Stream Data	Bit Stream Data	Bit Stream Data	Bit Stream Data	Bit Stream Data	Bit Stream Data	Bit Stream Data	Bit Stream Data	...

9.8.1 Huffman Codebook

The codebook is stored in such a way that it resembles the Huffman tree. Each entry in the codebook is either a branch or a leaf colour. The codebook should be interpreted as an array of 16 bit signed integers in little-endian format. A simple test can determine if a codebook entry w_n is a branch or a colour;

If $w_n \geq 0$ then w_n is a colour index from the main palette.

Else w_n is a branch.

Codebook branches are essentially a relative 'goto' instruction. The branch entry contains the destination address relative to the current position in the code book. Codebook branches use the following formula:

$$\text{Relative Jump} = \neg w_n \times 2 + 4$$

Instead of simply storing the palette entry in the leaves of the Huffman tree, a pair of values are stored:

Offset	Data Type	Content
0x00	Byte	Palette Entry
0x01	Byte	Size for Run Length Encoding (see section 9.8.2)

9.8.2 Run Length Encoding

The size field controls the run length encoding. It specifies how many bits of the bit stream to consume to obtain the count for the number of times to output that pixel entry. To ensure an optimal representation, an offset is added to the value read to ensure that no count can be represented in more than one way (which introduces redundancy).

Size	Number of extra bits to consume	Offset	Interval represented
0	0	1	[1, 2]
1	2	2	[2, 6]
2	4	6	[6, 22]
3	8	22	[22, 278]

Thus, this scheme can compress runs of up to 277 instances of the same value. Bits are read most significant bit first to construct the count.

The bits for Run Length Encoding are interlaced with the bits for Huffman decompression. Firstly the bits are consumed to find the correct leaf in the Huffman tree, then the relevant number of bits are consumed for Run Length Encoding.

9.8.3 Decompression Example

To decode the following compressed data;

03	00	00	00	FD	FF	FF	FF	54	03	34	02	1D	00	CC	CC	35	F2	AB

Split the data in each section using the size fields;

Size				Code Book Data												Bit Stream		
03	00	00	00	FD	FF	FF	FF	54	03	34	02	1D	00	CC	CC	35	F2	AB

The bytes with value 0xCC are padding to align the bit stream to a 4 byte boundary any may be any value.

Example sub-palette mapping;

Palette Index	**Example Colour**
0x54	Red
0x34	Green
0x1D	Blue

Refer to *figure 4* for a diagrammatic representation of the codebook.

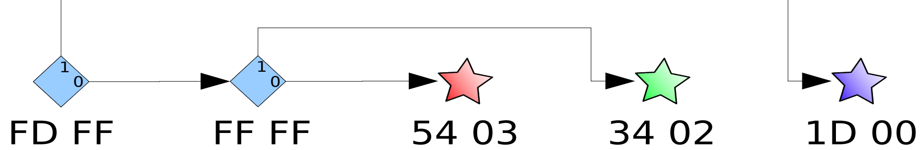

Figure 4: Diagrammatic Representation of Huffman Codebook

From section 9.8.1:

> 0xFFFD decodes as a relative jump forwards of 8 bytes.
>
> 0xFFFF decodes as a relative jump forwards of 4 bytes.

The bit stream can then be decompressed by processing one bit at a time until each colour has been decoded.

Using the example bit stream data;

Bit Stream Data
35 F2 AB 06 78 E4 ...

Expanding the bit stream (from left to right);

0 0 1 1 0 1 0 1	1 1 1 1 0 0 1 0	1 0 1 0 1 0 1 1

Decoded colours

0 0 1 1 0 1 0 1 1 1	1	1	0 0 1 0 1 0 1 0 1 0	1	1

The first leaf is red, and so we must read 8 bits for the count. 11010111b equals 215, so adding 22 gives 237 red pixels.

Following on are 2 blue pixels.

Then another red leaf, and so again we must read 8 bits for the count. 10101010b equals 170, so adding 22 gives 192 red pixels.

Following on are 2 blue pixels.

After all the data has been compressed, the tile must be de interlaced.

9.9 Digital Rights Management

QC3 files may have part of the compressed data stream encrypted. The only algorithm observed has been 128 bit AES in ECB mode. More interesting however is the algorithm to determine what is encrypted.

The third entry in the QC3 meta data header contains the level of encryption used. It may range of the value -1 up to 9. This value controls at what scale encryption starts. For example, a value of -1 indicates no encryption, and a value of 3 means $\frac{1}{16}$ of the rows are available.

To decrypt a tile, add 1 to the value in the header, and look up the scale size for that index. Start decrypting from that offset (multiplied by 4 as usual) to the end of the tile data.

This means that encrypted tiles can be successfully decoded up to the scale that encryption starts without knowing the key.