Theora Specification

Xiph.org Foundation

June 11, 2008

Contents

List of Figures

List of Tables

Notation and Conventions

All parameters either passed in or out of a decoding procedure are given in **bold face**.

The prefix `b` indicates that the following value is to be interpreted as a binary number (base 2).

Example: The value `b1110100` is equal to the decimal value 116.

The prefix `0x` indicates the the following value is to be interpreted as a hexadecimal number (base 16).

Example: The value `0x74` is equal to the decimal value 116.

All arithmetic defined by this specification is exact. However, any real numbers that do arise will always be converted back to integers again in short order. The entire specification can be implemented using only normal integer operations. All operations are to be implemented with sufficiently large integers so that overflow cannot occur. Where the result of a computation is to be truncated to a fixed-sized binary representation, this will be explicitly noted. The size given for all variables is the maximum number of bits needed to store any value in that variable. Intermediate computations involving that variable may require more bits.

The following operators are defined:

$|a|$ The absolute value of a number a.

$$|a| = \begin{cases} -a, & a < 0 \\ a, & a \geq 0 \end{cases}$$

$a * b$ Multiplication of a number a by a number b.

$\frac{a}{b}$ Exact division of a number a by a number b, producing a potentially non-integer result.

$\lfloor a \rfloor$ The largest integer less than or equal to a real number a.

$\lceil a \rceil$ The smallest integer greater than or equal to a real number a.

$a//b$ Integer division of a by b.

$$a//b = \begin{cases} \lceil \frac{a}{b} \rceil, & a < 0 \\ \lfloor \frac{a}{b} \rfloor, & a \geq 0 \end{cases}$$

$a\%b$ The remainder from the integer division of a by b.

$$a\%b = a - |b| * \left\lfloor \frac{a}{|b|} \right\rfloor$$

Note that with this definition, the result is always non-negative and less than $|b|$.

$a << b$ The value obtained by left-shifting the two's complement integer a by b bits. For purposes of this specification, overflow is ignored, and so this is equivalent to integer multiplication of a by 2^b.

$a >> b$ The value obtained by right-shifting the two's complement integer a by b bits, filling in the leftmost bits of the new value with 0 if a is non-negative and 1 if a is negative. This is *not* equivalent to integer division of a by 2^b. Instead,

$$a >> b = \left\lfloor \frac{a}{2^b} \right\rfloor.$$

round(a) Rounds a number a to the nearest integer, with ties rounded away from 0.

$$\text{round}(a) = \begin{cases} \lceil a - \frac{1}{2} \rceil & a \leq 0 \\ \lfloor a + \frac{1}{2} \rfloor & a > 0 \end{cases}$$

sign(a) Returns the sign of a given number.

$$\text{sign}(a) = \begin{cases} -1 & a < 0 \\ 0 & a = 0 \\ 1 & a > 0 \end{cases}$$

ilog(a) The minimum number of bits required to store a positive integer a in two's complement notation, or 0 for a non-positive integer a.

$$\text{ilog}(a) = \begin{cases} 0, & a \leq 0 \\ \lfloor \log_2 a \rfloor + 1, & a > 0 \end{cases}$$

Examples:

- $\text{ilog}(-1) = 0$
- $\text{ilog}(0) = 0$
- $\text{ilog}(1) = 1$
- $\text{ilog}(2) = 2$

- $\text{ilog}(3) = 2$
- $\text{ilog}(4) = 3$
- $\text{ilog}(7) = 3$

$\min(a, b)$ The minimum of two numbers a and b.

$\max(a, b)$ The maximum of two numbers a and b.

Key words

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be intrepreted as described in RFC 2119 [Bra97].

Where such assertions are placed on the contents of a Theora bitstream itself, implementations should be prepared to encounter bitstreams that do not follow these requirements. An application's behavior in the presecence of such non-conforming bitstreams is not defined by this specification, but any reasonable method of handling them MAY be used. By way of example, applications MAY discard the current frame, retain the current output thus far, or attempt to continue on by assuming some default values for the erroneous bits. When such an error occurs in the bitstream headers, an application MAY refuse to decode the entire stream. An application SHOULD NOT allow such non-conformant bitstreams to overflow buffers and potentially execute arbitrary code, as this represents a serious security risk.

An application MUST, however, ensure any bits marked as reserved have the value zero, and refuse to decode the stream if they do not. These are used as place holders for future bitstream features with which the current bitstream is forward-compatible. Such features may not increment the bitstream version number, and can only be recognized by checking the value of these reserved bits.

Chapter 1

Introduction

Theora is a general purpose, lossy video codec. It is based on the VP3 video codec produced by On2 Technologies (http://www.on2.com/). On2 donated the VP3.1 source code to the Xiph.org Foundation and released it under a BSD-like license. On2 also made an irrevocable, royalty-free license grant for any patent claims it might have over the software and any derivatives. No formal specification exists for the VP3 format beyond this source code, however Mike Melanson maintains a detailed description [Mel04]. Portions of this specification were adopted from that text with permission.

1.1 VP3 and Theora

Theora contains a superset of the features that were available in the original VP3 codec. Content encoded with VP3.1 can be losslessly transcoded into the Theora format. Theora content cannot, in general, be losslessly transcoded into the VP3 format. If a feature is not available in the original VP3 format, this is mentioned when that feature is defined. A complete list of these features appears in Appendix B.1.

1.2 Video Formats

Theora currently supports progressive video data of arbitrary dimensions at a constant frame rate in one of several $Y'C_bC_r$ color spaces. The precise definition the supported color spaces appears in Section 4.3. Three different chroma subsampling formats are supported: 4:2:0, 4:2:2, and 4:4:4. The precise details of each of these formats and their sampling locations are described in Section 4.4.

The Theora format does not support interlaced material, variable frame rates, bit-depths larger than 8 bits per component, nor alternate color spaces such as RGB or arbitrary multi-channel spaces. Black and white content can be efficiently encoded, however, because the uniform chroma planes compress well. Support for interlaced material is planned for a future version.

> **Note:** Infrequently changing frame rates—as when film and video sequences are cut together—can be supported in the Ogg container format by chaining several Theora streams together.

Support for increased bit depths or additional color spaces is not planned.

1.3 Classification

Theora is a block-based lossy transform codec that utilizes an 8×8 Type-II Discrete Cosine Transform and block-based motion compensation. This places it in the same class of codecs as MPEG-1, -2, -4, and H.263. The details of how individual blocks are organized and how DCT coefficients are stored in the bitstream differ substantially from these codecs, however. Theora supports only intra frames (I frames in MPEG) and inter frames (P frames in MPEG). There is no equivalent to the bi-predictive frames (B frames) found in MPEG codecs.

1.4 Assumptions

The Theora codec design assumes a complex, psychovisually-aware encoder and a simple, low-complexity decoder.

Theora provides none of its own framing, synchronization, or protection against transmission errors. An encoder is solely a method of accepting input video frames and compressing these frames into raw, unformatted 'packets'. The decoder then accepts these raw packets in sequence, decodes them, and synthesizes a fascimile of the original video frames. Theora is a free-form variable bit rate (VBR) codec, and packets have no minimum size, maximum size, or fixed/expected size.

Theora packets are thus intended to be used with a transport mechanism that provides free-form framing, synchronization, positioning, and error correction in accordance with these design assumptions, such as Ogg (for file transport) or RTP (for network multicast). For the purposes of a few examples in this document, we will assume that Theora is embedded in an Ogg stream specifically, although this is by no means a requirement or fundamental assumption in the Theora design.

The specification for embedding Theora into an Ogg transport stream is given in Appendix A.

1.5 Codec Setup and Probability Model

Theora's heritage is the proprietary commerical codec VP3, and it retains a fair amount of inflexibility when compared to Vorbis [Xip02], the first Xiph.org codec, which began as a research codec. However, to provide additional scope for encoder improvement, Theora adopts some of the configurable aspects of

decoder setup that are present in Vorbis. This configuration data is not available in VP3, which uses hardcoded values instead.

Theora makes the same controversial design decision that Vorbis made to include the entire probability model for the DCT coefficients and all the quantization parameters in the bitstream headers. This is often several hundred fields. It is therefore impossible to decode any frame in the stream without having previously fetched the codec info and codec setup headers.

> **Note:** Theora *can* initiate decode at an arbitrary intra-frame packet within a bitstream so long as the codec has been initialized with the setup headers.

Thus, Theora headers are both required for decode to begin and relatively large as bitstream headers go. The header size is unbounded, although as a rule-of-thumb less than 16kB is recommended, and Xiph.org's reference encoder follows this suggestion.

Our own design work indicates that the primary liability of the required header is in mindshare; it is an unusual design and thus causes some amount of complaint among engineers as this runs against current design trends and points out limitations in some existing software/interface designs. However, we find that it does not fundamentally limit Theora's suitable application space.

1.6 Format Conformance

The Theora format is well-defined by its decode specification; any encoder that produces packets that are correctly decoded by an implementation following this specification may be considered a proper Theora encoder. A decoder must faithfully and completely implement the specification defined herein to be considered a conformant Theora decoder. A decoder need not be implemented strictly as described, but the actual decoder process MUST be *entirely mathematically equivalent* to the described process. Where appropriate, a non-normative description of encoder processes is included. These sections will be marked as such, and a proper Theora encoder is not bound to follow them.

Chapter 2

Coded Video Structure

Theora's encoding and decoding process is based on 8×8 blocks of pixels. This sections describes how a video frame is laid out, divided into blocks, and how those blocks are organized.

2.1 Frame Layout

A video frame in Theora is a two-dimensional array of pixels. Theora, like VP3, uses a right-handed coordinate system, with the origin in the lower-left corner of the frame. This is contrary to many video formats which use a left-handed coordinate system with the origin in the upper-left corner of the frame.

Theora divides the pixel array up into three separate *color planes*, one for each of the Y', C_b, and C_r components of the pixel. The Y' plane is also called the *luma plane*, and the C_b and C_r planes are also called the *chroma planes*. Each plane is assigned a numerical value, as shown in Table 2.1.

Index	Color Plane
0	Y'
1	C_b
2	C_r

Table 2.1: Color Plane Indices

In some pixel formats, the chroma planes are subsampled by a factor of two in one or both directions. This means that the width or height of the chroma planes may be half that of the total frame width and height. The luma plane is never subsampled.

2.2 Picture Region

An encoded video frame in Theora is required to have a width and height that are multiples of sixteen, making an integral number of blocks even when the chroma planes are subsampled. However, inside a frame a smaller *picture region* may be defined to present material whose dimensions are not a multiple of sixteen pixels, as shown in Figure 2.1. The picture region can be offset from the lower-left corner of the frame by up to 255 pixels in each direction, and may have an arbitrary width and height, provided that it is contained entirely within the coded frame. It is this picture region that contains the actual video data. The portions of the frame which lie outside the picture region may contain arbitrary image data, so the frame must be cropped to the picture region before display. The picture region plays no other role in the decode process, which operates on the entire video frame.

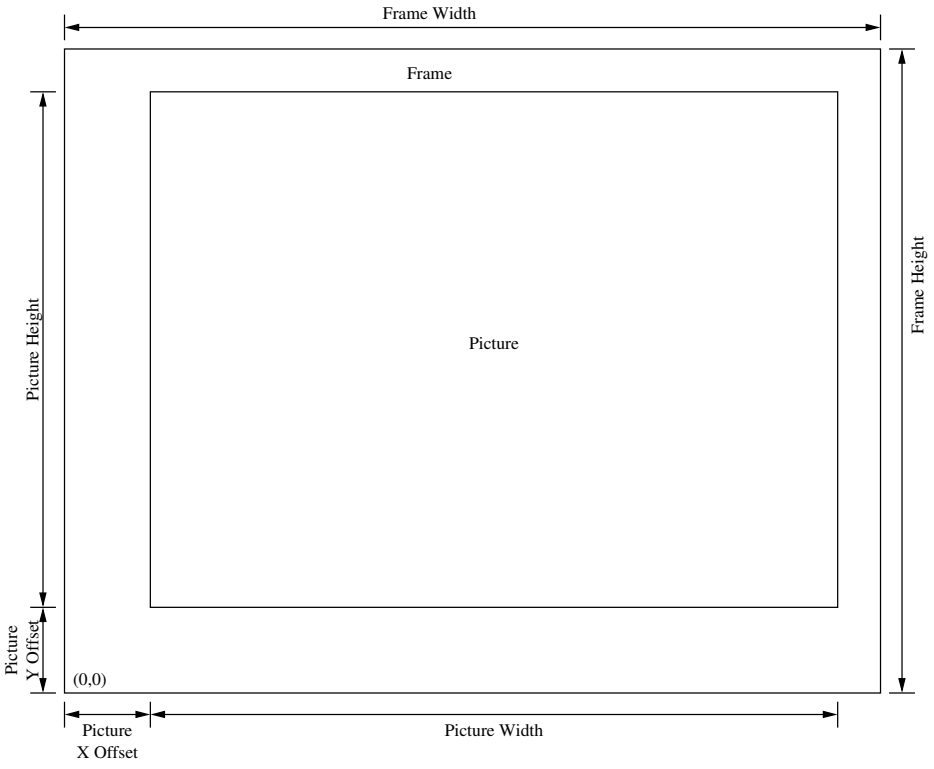


Figure 2.1: Location of frame and picture regions

2.3 Blocks and Super Blocks

Each color plane is subdivided into *blocks* of 8×8 pixels. Blocks are grouped into 4×4 arrays called *super blocks* as shown in Figure 2.2. Each color plane has its own set of blocks and super blocks. If the chroma planes are subsampled, they are still divided into 8×8 blocks of pixels; there are just fewer blocks than in the luma plane. The boundaries of blocks and super blocks in the luma plane do not necessarily coincide with those of the chroma planes, if the chroma planes have been subsampled.

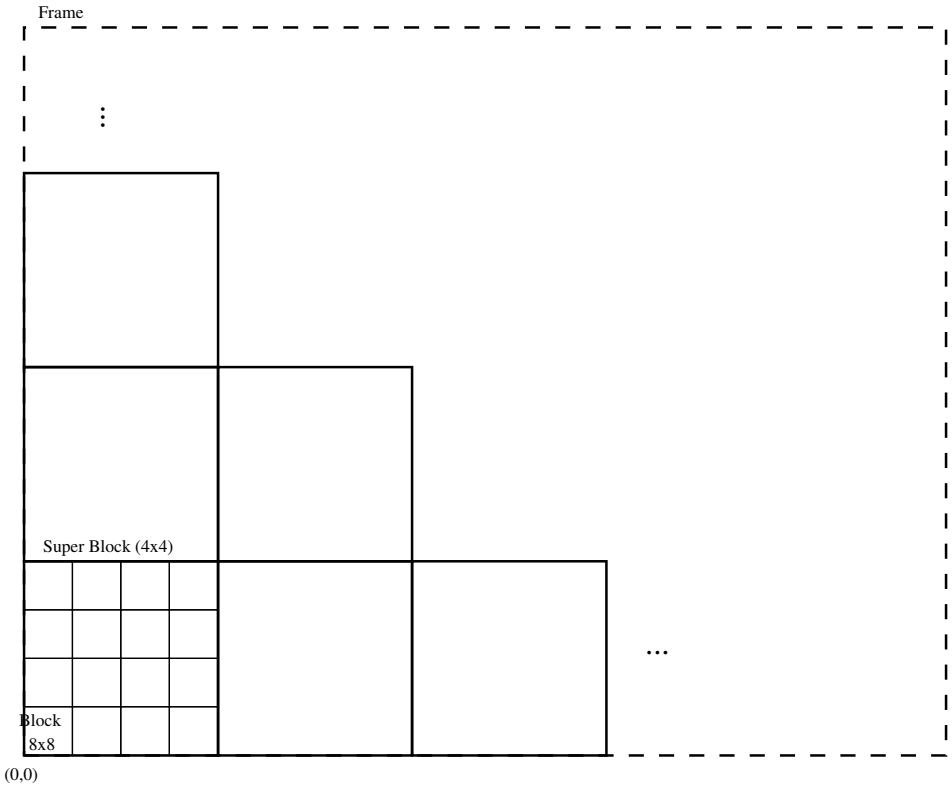


Figure 2.2: Subdivision of a frame into blocks and super blocks

Blocks are accessed in two different orders in the various decoder processes. The first is *raster order*, illustrated in Figure 2.3. This accesses each block in row-major order, starting in the lower left of the frame and continuing along the bottom row of the entire frame, followed by the next row up, starting on the left edge of the frame, etc.

The second is *coded order*. In coded order, blocks are accessed by super block. Within each frame, super blocks are traversed in raster order, similar to

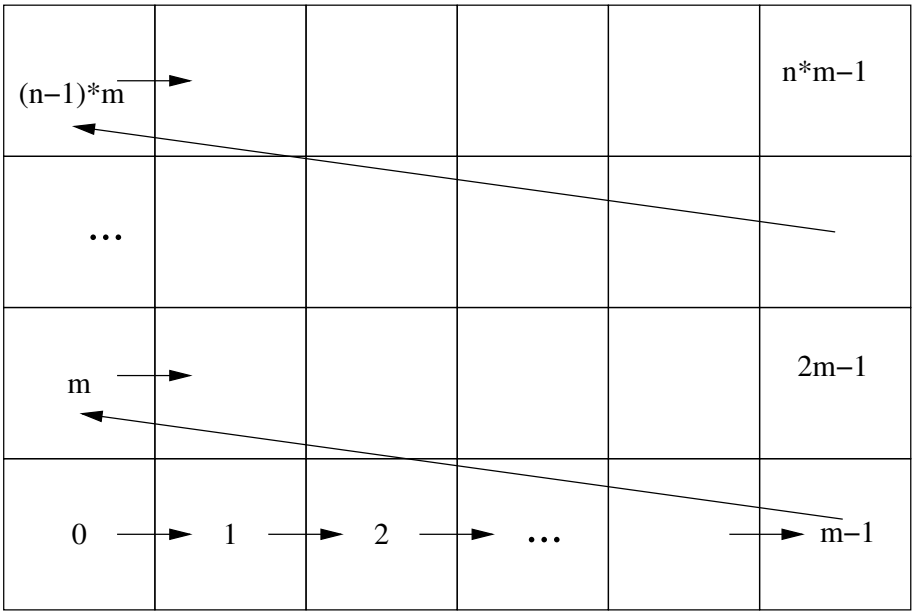


Figure 2.3: Raster ordering of $n \times m$ blocks

raster order for blocks. Within each super block, however, blocks are accessed in a Hilbert curve pattern, illustrated in Figure 2.4. If a color plane does not contain a complete super block on the top or right sides, the same ordering is still used, simply with any blocks outside the frame boundary ommitted.

To illustrate this ordering, consider a frame that is 240 pixels wide and 48 pixels high. Each row of the luma plane has 30 blocks and 8 super blocks, and there are 6 rows of blocks and two rows of super blocks.

When accessed in coded order, each block in the luma plane is assigned the following indices:

123	122	125	124	...	179	178
120	121	126	127	...	176	177
5	6	9	10	...	117	118
4	7	8	11	...	116	119
3	2	13	12	...	115	114
0	1	14	15	...	112	113

Here the index values specify the order in which the blocks would be accessed. The indices of the blocks are numbered continuously from one color plane to the next. They do not reset to zero at the start of each plane. Instead, the numbering increases continuously from the Y' plane to the C_b plane to the C_r

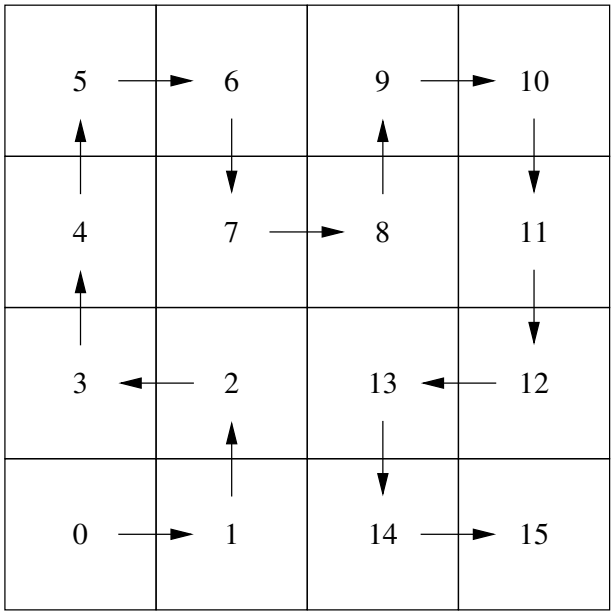


Figure 2.4: Hilbert curve ordering of blocks within a super block

plane. The implication is that the blocks from all planes are treated as a unit during the various processing steps.

Although blocks are sometimes accessed in raster order, in this document the index associated with a block is *always* its index in coded order.

2.4 Macro Blocks

A macro block contains a 2×2 array of blocks in the luma plane *and* the co-located blocks in the chroma planes, as shown in Figure 2.5. Thus macro blocks can represent anywhere from six to twelve blocks, depending on how the chroma planes are subsampled. This is in contrast to super blocks, which only contain blocks from a single color plane. Macro blocks contain information about coding mode and motion vectors for the corresponding blocks in all color planes.

Macro blocks are also accessed in a *coded order*. This coded order proceeds by examining each super block in the luma plane in raster order, and traversing the four macro blocks inside using a smaller Hilbert curve, as shown in Figure 2.6. If the luma plane does not contain a complete super block on the top or right sides, the same ordering is still used, with any macro blocks outside the frame boundary simply omitted. Because the frame size is constrained to be a multiple of 16, there are never any partial macro blocks. Unlike blocks, macro blocks need never be accessed in a pure raster order.

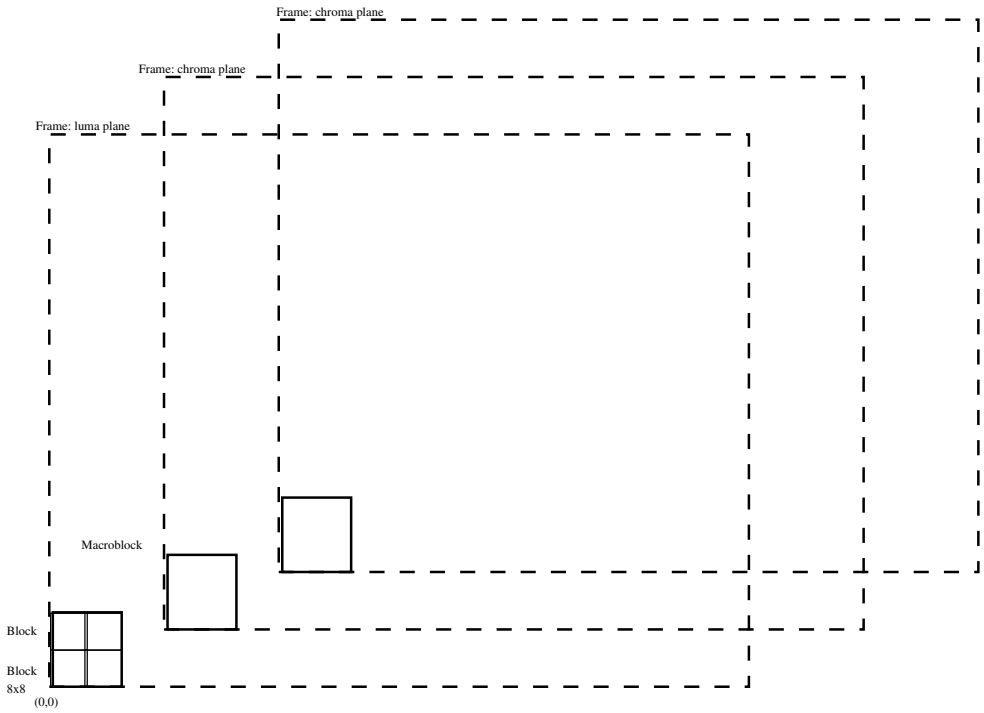


Figure 2.5: Subdivision of a frame into macro blocks

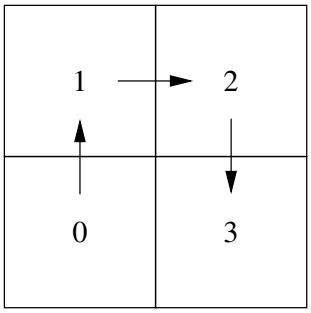


Figure 2.6: Hilbert curve ordering of macro blocks within a super block

Using the same frame size as the example above, there are 15 macro blocks in each row and 3 rows of macro blocks. The macro blocks are assigned the following indices:

30	31	32	33	···	42	43	44
1	2	5	6	···	25	26	29
0	3	4	7	···	24	27	28

2.5 Coding Modes and Prediction

Each block is coded using one of a small, fixed set of *coding modes* that define how the block is predicted from previous frames. A block is predicted using one of two *reference frames*, selected according to the coding mode. A reference frame is the fully decoded version of a previous frame in the stream. The first available reference frame is the previous intra frame, called the *golden frame*. The second available reference frame is the previous frame, whether it was an intra frame or an inter frame. If the previous frame was an intra frame, then both reference frames are the same. See Figure 2.7 for an illustration of the reference frames used for an intra frame that does not follow an intra frame.

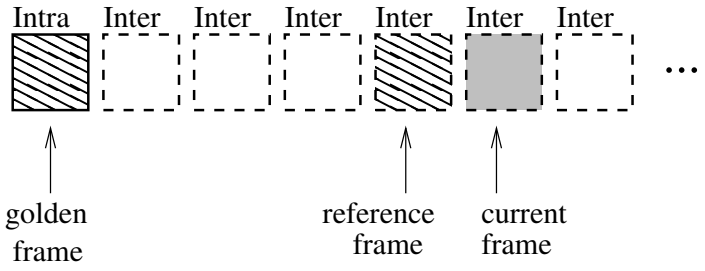


Figure 2.7: Example of reference frames for an inter frame

Two coding modes in particular are worth mentioning here. The INTRA mode is used for blocks that are not predicted from either reference frame. This is the only coding mode allowed in intra frames. The INTER_NOMV coding mode uses the co-located contents of the block in the previous frame as the predictor. This is the default coding mode.

2.6 DCT Coefficients

A *residual* is added to the predicted contents of a block to form the final reconstruction. The residual is stored as a set of quantized coefficients from an

integer approximation of a two-dimensional Type II Discrete Cosine Transform. The DCT takes an 8×8 array of pixel values as input and returns an 8×8 array of coefficient values. The *natural ordering* of these coefficients is defined to be row-major order, from lowest to highest frequency. They are also often indexed in *zig-zag order*, as shown in Figure 2.8.

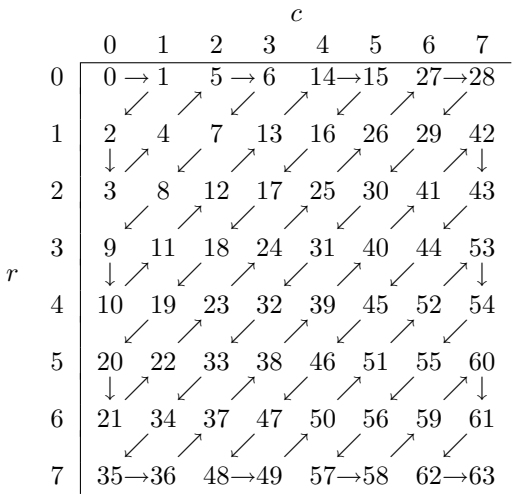


Figure 2.8: Zig-zag order

Note: the row and column indices refer to *frequency number* and not pixel locations. The frequency numbers are defined independently of the memory organization of the pixels. They have been written from top to bottom here to follow conventional notation, despite the right-handed coordinate system Theora uses for pixel locations. Many implementations of the DCT operate 'in-place'. That is, they return DCT coefficients in the same memory buffer that the initial pixel values were stored in. Due to the right-handed coordinate system used for pixel locations in Theora, one must note carefully how both pixel values and DCT coefficients are organized in memory in such a system.

DCT coefficient $(0, 0)$ is called the *DC coefficient*. All the other coefficients are called *AC coefficients*.

Chapter 3

Decoding Overview

This section provides a high level description of the Theora codec's construction. A bit-by-bit specification appears beginning in Section 5. The later sections assume a high-level understanding of the Theora decode process, which is provided below.

3.1 Decoder Configuration

Decoder setup consists of configuration of the quantization matrices and the Huffman codebooks for the DCT coefficients, and a table of limit values for the deblocking filter. The remainder of the decoding pipeline is not configurable.

3.1.1 Global Configuration

The global codec configuration consists of a few video related fields, such as frame rate, frame size, picture size and offset, aspect ratio, color space, pixel format, and a version number. The version number is divided into a major version, a minor version, amd a minor revision number. For the format defined in this specification, these are '3', '2', and '1', respectively, in reference to Theora's origin as a successor to the VP3.1 format.

3.1.2 Quantization Matrices

Theora allows up to 384 different quantization matrices to be defined, one for each *quantization type*, *color plane* (Y', C_b, or C_r), and *quantization index*, *qi*, which ranges from zero to 63, inclusive. There are currently two quantization types defined, which depend on the coding mode of the block being dequantized, as shown in Table 3.1.

The quantization index, on the other hand, nominally represents a progressive range of quality levels, from low quality near zero to high quality near 63. However, the interpretation is arbitrary, and it is possible, for example, to partition the scale into two completely separate ranges with 32 levels each that are

Quantization Type	Usage
0	INTRA-mode blocks
1	Blocks in any other mode.

Table 3.1: Quantization Type Indices

meant to represent different classes of source material, or any other arrangement that suits the encoder's requirements.

Each quantization matrix is an 8×8 matrix of 16-bit values, which is used to quantize the output of the 8×8 DCT. Quantization matrices are specified using three components: a *base matrix* and two *scale values*. The first scale value is the *DC scale*, which is applied to the DC component of the base matrix. The second scale value is the *AC scale*, which is applied to all the other components of the base matrix. There are 64 DC scale values and 64 AC scale values, one for each *qi* value.

There are 64 elements in each base matrix, one for each DCT coefficient. They are stored in natural order (cf. Section 2.6). There is a separate set of base matrices for each quantization type and each color plane, with up to 64 possible base matrices in each set, one for each *qi* value. Typically the bitstream contains matrices for only a sparse subset of the possible *qi* values. The base matrices for the remainder of the *qi* values are computed using linear interpolation. This configuration allows the encoder to adjust the quantization matrices to approximate the complex, non-linear response of the human visual system to different quantization errors.

Finally, because the in-loop deblocking filter strength depends on the strength of the quantization matrices defined in this header, a table of 64 *loop filter limit values* is defined, one for each *qi* value.

The precise specification of how all of this information is decoded appears in Section 6.4.1 and Section 6.4.2.

3.1.3 Huffman Codebooks

Theora uses 80 configurable binary Huffman codes to represent the 32 tokens used to encode DCT coefficients. Each of the 32 token values has a different semantic meaning and is used to represent single coefficient values, zero runs, combinations of the two, and *End-Of-Block markers*.

The 80 codes are divided up into five groups of 16, with each group corresponding to a set of DCT coefficient indices. The first group corresponds to the DC coefficient, while the remaining four groups correspond to different subsets of the AC coefficients. Within each frame, two pairs of 4-bit codebook indices are stored. The first pair selects which codebooks to use from the DC coefficient group for the Y' coefficients and the C_b and C_r coefficients. The second pair selects which codebooks to use from *all four* of the AC coefficient groups for the Y' coefficients and the C_b and C_r coefficients.

The precise specification of how the codebooks are decoded appears in Section 6.4.4.

3.2 High-Level Decode Process

3.2.1 Decoder Setup

Before decoding can begin, a decoder MUST be initialized using the bitstream headers corresponding to the stream to be decoded. Theora uses three header packets; all are required, in order, by this specification. Once set up, decode may begin at any intra-frame packet—or even inter-frame packets, provided the appropriate decoded reference frames have already been decoded and cached—belonging to the Theora stream. In Theora I, all packets after the three initial headers are intra-frame or inter-frame packets.

The header packets are, in order, the identification header, the comment header, and the setup header.

Identification Header The identification header identifies the stream as Theora, provides a version number, and defines the characteristics of the video stream such as frame size. A complete description of the identification header appears in Section 6.2.

Comment Header The comment header includes user text comments ('tags') and a vendor string for the application/library that produced the stream. The format of the comment header is the same as that used in the Vorbis I and Speex codecs, with slight modifications due to the use of a different bit packing mechanism. A complete description of how the comment header is coded appears in Section 6.3, along with a suggested set of tags.

Setup Header The setup header includes extensive codec setup information, including the complete set of quantization matrices and Huffman codebooks needed to decode the DCT coefficients. A complete description of the setup header appears in Section 6.4.

3.2.2 Decode Procedure

The decoding and synthesis procedure for all video packets is fundamentally the same, with some steps omitted for intra frames.

- Decode packet type flag.
- Decode frame header.
- Decode coded block information (inter frames only).
- Decode macro block mode information (inter frames only).

- Decode motion vectors (inter frames only).
- Decode block-level qi information.
- Decode DC coefficient for each coded block.
- Decode 1st AC coefficient for each coded block.
- Decode 2nd AC coefficient for each coded block.
- ...
- Decode 63rd AC coefficient for each coded block.
- Perform DC coefficient prediction.
- Reconstruct coded blocks.
- Copy uncoded bocks.
- Perform loop filtering.

  Note: clever rearrangement of the steps in this process is possible. As an example, in a memory-constrained environment, one can make multiple passes through the DCT coefficients to avoid buffering them all in memory. On the first pass, the starting location of each coefficient is identified, and then 64 separate get pointers are used to read in the 64 DCT coefficients required to reconstruct each coded block in sequence. This operation produces entirely equivalent output and is naturally perfectly legal. It may even be a benefit in non-memory-constrained environments due to a reduced cache footprint.

Theora makes equivalence easy to check by defining all decoding operations in terms of exact integer operations. No floating-point math is required, and in particular, the implementation of the iDCT transform MUST be followed precisely. This prevents the decoder mismatch problem commonly associated with codecs that provide a less rigorous transform specification. Such a mismatch problem would be devastating to Theora, since a single rounding error in one frame could propagate throughout the entire succeeding frame due to DC prediction.

Packet Type Decode Theora uses four packet types. The first three packet types mark each of the three Theora headers described above. The fourth packet type marks a video packet. All other packet types are reserved; packets marked with a reserved type should be ignored.

Additionally, zero-length packets are treated as if they were an inter frame with no blocks coded. That is, as a duplicate frame.

Frame Header Decode The frame header contains some global information about the current frame. The first is the frame type field, which specifies if this is an intra frame or an inter frame. Inter frames predict their contents from previously decoded reference frames. Intra frames can be independently decoded with no established reference frames.

The next piece of information in the frame header is the list of *qi* values allowed in the frame. Theora allows from one to three different *qi* values to be used in a single frame, each of which selects a set of six quantization matrices, one for each quantization type (inter or intra), and one for each color plane. The first *qi* value is *always* used when dequantizing DC coefficients. The *qi* value used when dequantizing AC coefficients, however, can vary from block to block. VP3, in contrast, only allows a single *qi* value per frame for both the DC and AC coefficients.

Coded Block Information This stage determines which blocks in the frame are coded and which are uncoded. A *coded block list* is constructed which lists all the coded blocks in coded order. For intra frames, every block is coded, and so no data needs to be read from the packet.

Macro Block Mode Information For intra frames, every block is coded in INTRA mode, and this stage is skipped. In inter frames a *coded macro block list* is constructed from the coded block list. Any macro block which has at least one of its luma blocks coded is considered coded; all other macro blocks are uncoded, even if they contain coded chroma blocks. A coding mode is decoded for each coded macro block, and assigned to all its constituent coded blocks. All coded chroma blocks in uncoded macro blocks are assigned the INTER_NOMV coding mode.

Motion Vectors Intra frames are coded entirely in INTRA mode, and so this stage is skipped. Some inter coding modes, however, require one or more motion vectors to be specified for each macro block. These are decoded in this stage, and an appropriate motion vector is assigned to each coded block in the macro block.

Block-Level *qi* Information If a frame allows multiple *qi* values, the *qi* value assigned to each block is decoded here. Frames that use only a single *qi* value have nothing to decode.

DCT Coefficients Finally, the quantized DCT coefficients are decoded. A list of DCT coefficients in zig-zag order for a single block is represented by a list of tokens. A token can take on one of 32 different values, each with a different semantic meaning. A single token can represent a single DCT coefficient, a run of zero coefficients within a single block, a combination of a run of zero coefficients followed by a single non-zero coefficient, an *End-Of-Block marker*, or a run of EOB markers. EOB markers signify that the remainder of the block

is one long zero run. Unlike JPEG and MPEG, there is no requirement for each block to end with a special marker. If non-EOB tokens yield values for all 64 of the coefficients in a block, then no EOB marker occurs.

Each token is associated with a specific *token index* in a block. For single-coefficient tokens, this index is the zig-zag index of the token in the block. For zero-run tokens, this index is the zig-zag index of the *first* coefficient in the run. For combination tokens, the index is again the zig-zag index of the first coefficient in the zero run. For EOB markers, which signify that the remainder of the block is one long zero run, the index is the zig-zag index of the first zero coefficient in that run. For EOB runs, the token index is that of the first EOB marker in the run. Due to zero runs and EOB markers, a block does not have to have a token for every zig-zag index.

Tokens are grouped in the stream by token index, not by the block they originate from. This means that for each zig-zag index in turn, the tokens with that index from *all* the coded blocks are coded in coded block order. When decoding, a current token index is maintained for each coded block. This index is advanced by the number of coefficients that are added to the block as each token is decoded. After fully decoding all the tokens with token index ti, the current token index of every coded block will be ti or greater.

If an EOB run of n blocks is decoded at token index ti, then it ends the next n blocks in coded block order whose current token index is equal to ti, but not greater. If there are fewer than n blocks with a current token index of ti, then the decoder goes through the coded block list again from the start, ending blocks with a current token index of $ti + 1$, and so on, until n blocks have been ended.

Tokens are read by parsing a Huffman code that depends on ti and the color plane of the next coded block whose current token index is equal to ti, but not greater. The Huffman codebooks are selected on a per-frame basis from the 80 codebooks defined in the setup header. Many tokens have a fixed number of *extra bits* associated with them. These bits are read from the packet immediately after the token is decoded. These are used to define things such as coefficient magnitude, sign, and the length of runs.

DC Prediction After the coefficients for each block are decoded, the quantized DC value of each block is adjusted based on the DC values of its neighbors. This adjustment is performed by scanning the blocks in raster order, not coded block order.

Reconstruction Finally, using the coding mode, motion vector (if applicable), quantized coefficient list, and qi value defined for each block, all the coded blocks are reconstructed. The DCT coefficients are dequantized, an inverse DCT transform is applied, and the predictor is formed from the coding mode and motion vector and added to the result.

Loop Filtering To complete the reconstructed frame, an "in-loop" deblocking filter is applied to the edges of all coded blocks.

Chapter 4

Video Formats

This section gives a precise description of the video formats that Theora is capable of storing. The Theora bitstream is capable of handling video at any arbitrary resolution up to 1048560×1048560. Such video would require almost three terabytes of storage per frame for uncompressed data, so compliant decoders MAY refuse to decode images with sizes beyond their capabilities.

The remainder of this section talks about two specific aspects of the video format: the color space and the pixel format. The first describes how color is represented and how to transform that color representation into a device independent color space such as CIE XYZ (1931). The second describes the various schemes for sampling the color values in time and space.

4.1 Color Space Conventions

There are a large number of different color standards used in digital video. Since Theora is a lossy codec, it restricts itself to only a few of them to simplify playback. Unlike the alternate method of describing all the parameters of the color model, this allows a few dedicated routines for color conversion to be written and heavily optimized in a decoder. More flexible conversion functions should instead be specified in an encoder, where additional computational complexity is more easily tolerated. The color spaces were selected to give a fair representation of color standards in use around the world today. Most of the standards that do not exactly match one of these can be converted to one fairly easily.

All Theora color spaces are $Y'C_bC_r$ color spaces with one luma channel and two chroma channels. Each channel contains 8-bit discrete values in the range $0 \ldots 255$, which represent non-linear gamma pre-corrected signals. The Theora identification header contains an 8-bit value that describes the color space. This merely selects one of the color spaces available from an enumerated list. Currently, only two color spaces are defined, with a third possibility that indicates the color space is "unknown".

4.2 Color Space Conversions and Parameters

The parameters which describe the conversions between each color space are listed below. These are the parameters needed to map colors from the encoded $Y'C_bC_r$ representation to the device-independent color space CIE XYZ (1931). These parameters define abstract mathematical conversion functions which are infinitely precise. The accuracy and precision with which the conversions are performed in a real system is determined by the quality of output desired and the available processing power. Exact decoder output is defined by this specification only in the original $Y'C_bC_r$ space.

$Y'C_bC_r$ to $Y'P_bP_r$:

This conversion takes 8-bit discrete values in the range $[0 \dots 255]$ and maps them to real values in the range $[0 \dots 1]$ for Y and $[-\frac{1}{2} \dots \frac{1}{2}]$ for P_b and P_r. Because some values may fall outside the offset and excursion defined for each channel in the $Y'C_bC_r$ space, the results may fall outside these ranges in $Y'P_bP_r$ space. No clamping should be done at this stage.

$$Y'_{\text{out}} = \frac{Y'_{\text{in}} - \text{Offset}_Y}{\text{Excursion}_Y} \tag{4.1}$$

$$P_b = \frac{C_b - \text{Offset}_{C_b}}{\text{Excursion}_{C_b}} \tag{4.2}$$

$$P_r = \frac{C_r - \text{Offset}_{C_r}}{\text{Excursion}_{C_r}} \tag{4.3}$$

Parameters: $\text{Offset}_{Y,C_b,C_r}$, $\text{Excursion}_{Y,C_b,C_r}$.

$Y'P_bP_r$ to $R'G'B'$:

This conversion takes the one luma and two chroma channel representation and maps it to the non-linear $R'G'B'$ space used to drive actual output devices. Values should be clamped into the range $[0 \dots 1]$ after this stage.

$$R' = Y' + 2(1 - K_r)P_r \tag{4.4}$$

$$G' = Y' - 2\frac{(1 - K_b)K_b}{1 - K_b - K_r}P_b - 2\frac{(1 - K_r)K_r}{1 - K_b - K_r}P_r \tag{4.5}$$

$$B' = Y' + 2(1 - K_b)P_b \tag{4.6}$$

Parameters: K_b, K_r.

$R'G'B'$ to RGB (Output device gamma correction):

This conversion takes the non-linear $R'G'B'$ voltage levels and maps them to linear light levels produced by the actual output device. Note that this conversion is only that of the output device, and its inverse is *not* that used by the input device. Because a dim viewing environment is assumed in most television standards, the overall gamma between the input and output devices is usually around 1.1 to 1.2, and not a strict 1.0.

For calibration with actual output devices, the model

$$L = (E' + \Delta)^{\gamma} \tag{4.7}$$

should be used, with Δ the free parameter and γ held fixed to the value specified in this document. The conversion function presented here is an idealized version with $\Delta = 0$.

$$R = R'^{\gamma} \tag{4.8}$$
$$G = G'^{\gamma} \tag{4.9}$$
$$B = B'^{\gamma} \tag{4.10}$$

Parameters: γ.

RGB **to** $R'G'B'$ **(Input device gamma correction):**

This conversion takes linear light levels and maps them to the non-linear voltage levels produced in the actual input device. This information is merely informative. It is not required for building a decoder or for converting between the various formats and the actual output capabilities of a particular device.

A linear segment is introduced on the low end to reduce noise in dark areas of the image. The rest of the scale is adjusted so that the power segment of the curve intersects the linear segment with the proper slope, and so that it still maps 0 to 0 and 1 to 1.

$$R' = \begin{cases} \alpha R, & 0 \le R < \delta \\ (1+\epsilon)R^{\beta} - \epsilon, & \delta \le R \le 1 \end{cases} \tag{4.11}$$
$$G' = \begin{cases} \alpha G, & 0 \le G < \delta \\ (1+\epsilon)G^{\beta} - \epsilon, & \delta \le G \le 1 \end{cases} \tag{4.12}$$
$$B' = \begin{cases} \alpha B, & 0 \le B < \delta \\ (1+\epsilon)B^{\beta} - \epsilon, & \delta \le B \le 1 \end{cases} \tag{4.13}$$

Parameters: β, α, δ, ϵ.

RGB **to CIE** *XYZ* **(1931):**

This conversion maps a device-dependent linear RGB space to the device-independent linear CIE XYZ space. The parameters are the CIE chromaticity coordinates of the three primaries—red, green, and blue—as well as the chromaticity coordinates of the white point of the device. This is how hardware manufacturers and standards typically describe a particular RGB space. The math required to convert these parameters into a useful transformation matrix is reproduced below.

$$F = \begin{bmatrix} \frac{x_r}{y_r} & \frac{x_g}{y_g} & \frac{x_b}{y_b} \\ 1 & 1 & 1 \\ \frac{1-x_r-y_r}{y_r} & \frac{1-x_g-y_g}{y_g} & \frac{1-x_b-y_b}{y_b} \end{bmatrix} \tag{4.14}$$

$$\begin{bmatrix} s_r \\ s_g \\ s_b \end{bmatrix} = F^{-1} \begin{bmatrix} \frac{x_w}{y_w} \\ 1 \\ \frac{1-x_w-y_w}{y_w} \end{bmatrix} \tag{4.15}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = F \begin{bmatrix} s_r R \\ s_g G \\ s_b B \end{bmatrix} \tag{4.16}$$

Parameters: $x_r, x_g, x_b, x_w, y_r, y_g, y_b, y_w$.

4.3 Available Color Spaces

These are the color spaces currently defined for use by Theora video. Each one has a short name, with which it is referred to in this document, and a more detailed specification of the standards from which its parameters are derived. Some standards do not specify all the parameters necessary. For these unspecified parameters, this document serves as the definition of what should be used when encoding or decoding Theora video.

4.3.1 Rec. 470M (Rec. ITU-R BT.470-6 System M/NTSC with Rec. ITU-R BT.601-5)

This color space is used by broadcast television and DVDs in much of the Americas, Japan, Korea, and the Union of Myanmar [ITU98]. This color space may also be used for System M/PAL (Brazil), with an appropriate conversion supplied by the encoder to compensate for the different gamma value. See Section 4.3.2 for an appropriate gamma value to assume for M/PAL input.

In the US, studio monitors are adjusted to a D65 white point ($x_w, y_w = 0.313, 0.329$). In Japan, studio monitors are adjusted to a D white of 9300K ($x_w, y_w = 0.285, 0.293$).

Rec. 470 does not specify a digital encoding of the color signals. For Theora, Rec. ITU-R BT.601-5 [ITU95] is used, starting from the $R'G'B'$ signals specified by Rec. 470.

Rec. 470 does not specify an input gamma function. For Theora, the Rec. 709 [ITU02] input function is assumed. This is the same as that specified by SMPTE 170M [SMP94], which claims to reflect modern practice in the creation of NTSC signals circa 1994.

The parameters for all the color transformations defined in Section 4.2 are given in Table 4.1.

$$
\begin{aligned}
\text{Offset}_{Y,C_b,C_r} &= (16, 128, 128) \\
\text{Excursion}_{Y,C_b,C_r} &= (219, 224, 224) \\
K_r &= 0.299 \\
K_b &= 0.114 \\
\gamma &= 2.2 \\
\beta &= 0.45 \\
\alpha &= 4.5 \\
\delta &= 0.018 \\
\epsilon &= 0.099 \\
x_r, y_r &= 0.67, 0.33 \\
x_g, y_g &= 0.21, 0.71 \\
x_b, y_b &= 0.14, 0.08 \\
\text{(Illuminant C) } x_w, y_w &= 0.310, 0.316
\end{aligned}
$$

Table 4.1: Rec. 470M Parameters

4.3.2 Rec. 470BG (Rec. ITU-R BT.470-6 Systems B and G with Rec. ITU-R BT.601-5)

This color space is used by the PAL and SECAM systems in much of the rest of the world [ITU98] This can be used directly by systems (B, B1, D, D1, G, H, I, K, N)/PAL and (B, D, G, H, K, K1, L)/SECAM.

> **Note:** the Rec. 470BG chromaticity values are different from those specified in Rec. 470M. When PAL and SECAM systems were first designed, they were based upon the same primaries as NTSC. However, as methods of making color picture tubes have changed, the primaries used have changed as well. The U.S. recommends using correction circuitry to approximate the existing, standard NTSC primaries. Current PAL and SECAM systems have standardized on primaries in accord with more recent technology.

Rec. 470 provisionally permits the use of the NTSC chromaticity values (given in Section 4.3.1) with legacy PAL and SECAM equipment. In Theora, material must be decoded assuming the new PAL and SECAM primaries. Material intended for display on old legacy devices should be converted by the decoder.

The official Rec. 470BG specifies a gamma value of $\gamma = 2.8$. However, in practice this value is unrealistically high [Poy97]. Rec. 470BG states that the overall system gamma should be approximately $\gamma\beta = 1.2$. Since most cameras pre-correct with a gamma value of $\beta = 0.45$, this suggests an output device gamma of approximately $\gamma = 2.67$. This is the value recommended for use with PAL systems in Theora.

Rec. 470 does not specify a digital encoding of the color signals. For Theora, Rec. ITU-R BT.601-5 [ITU95] is used, starting from the $R'G'B'$ signals specified by Rec. 470.

Rec. 470 does not specify an input gamma function. For Theora, the Rec 709 [ITU02] input function is assumed.

The parameters for all the color transformations defined in Section 4.2 are given in Table 4.2.

$$
\begin{aligned}
\text{Offset}_{Y,C_b,C_r} &= (16, 128, 128) \\
\text{Excursion}_{Y,C_b,C_r} &= (219, 224, 224) \\
K_r &= 0.299 \\
K_b &= 0.114 \\
\gamma &= 2.67 \\
\beta &= 0.45 \\
\alpha &= 4.5 \\
\delta &= 0.018 \\
\epsilon &= 0.099 \\
x_r, y_r &= 0.64, 0.33 \\
x_g, y_g &= 0.29, 0.60 \\
x_b, y_b &= 0.15, 0.06 \\
\text{(D65)}\ x_w, y_w &= 0.313, 0.329
\end{aligned}
$$

Table 4.2: Rec. 470BG Parameters

4.4 Pixel Formats

Theora supports several different pixel formats, each of which uses different subsampling for the chroma planes relative to the luma plane. A decoder may

need to recover a full resolution chroma plane with samples co-sited with the luma plane in order to convert to RGB for display or perform other processing. Decoders can assume that the chroma signal satisfies the Nyquist-Shannon sampling theorem. The ideal low-pass reconstruction filter this implies is not practical, but any suitable approximation can be used, depending on the available computing power. Decoders MAY simply use a box filter, assigning to each luma sample the chroma sample closest to it. Encoders would not go wrong in assuming that this will be the most common approach.

4.4.1 4:4:4 Subsampling

All three color planes are stored at full resolution—each pixel has a Y', a C_b and a C_r value (see Figure 4.1). The samples in the different planes are all at co-located sites.

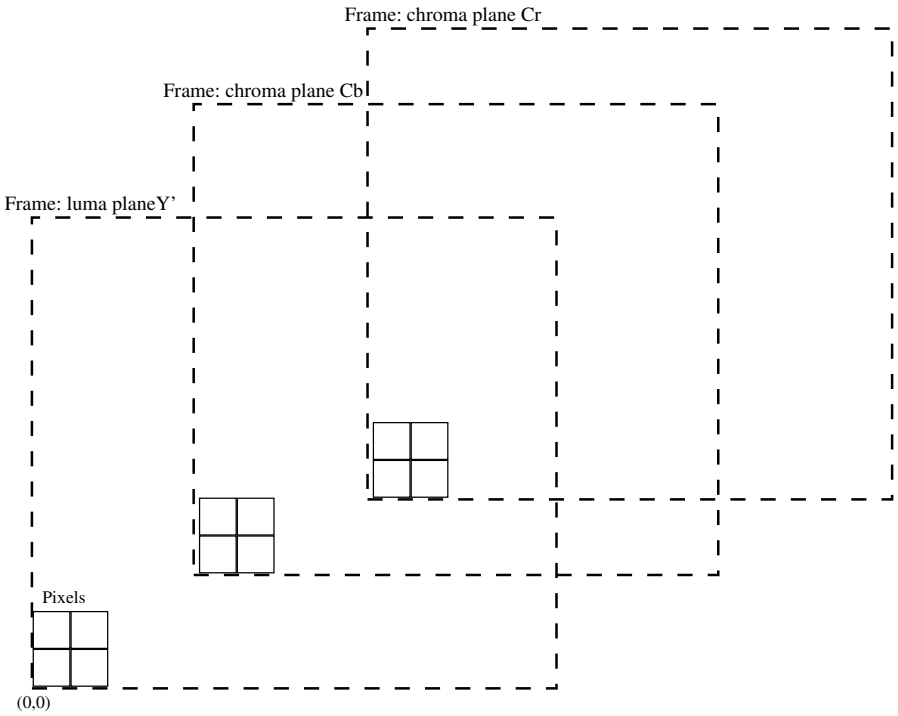


Figure 4.1: Pixels encoded 4:4:4

4.4.2 4:2:2 Subsampling

The C_b and C_r planes are stored with half the horizontal resolution of the Y' plane. Thus, each of these planes has half the number of horizontal blocks as the

luma plane (see Figure 4.2). Similarly, they have half the number of horizontal super blocks, rounded up. Macro blocks are defined across color planes, and so their number does not change, but each macro block contains half as many chroma blocks.

The chroma samples are vertically aligned with the luma samples, but horizontally centered between two luma samples. Thus, each luma sample has a unique closest chroma sample. A horizontal phase shift may be required to produce signals which use different horizontal chroma sampling locations for compatibility with different systems.

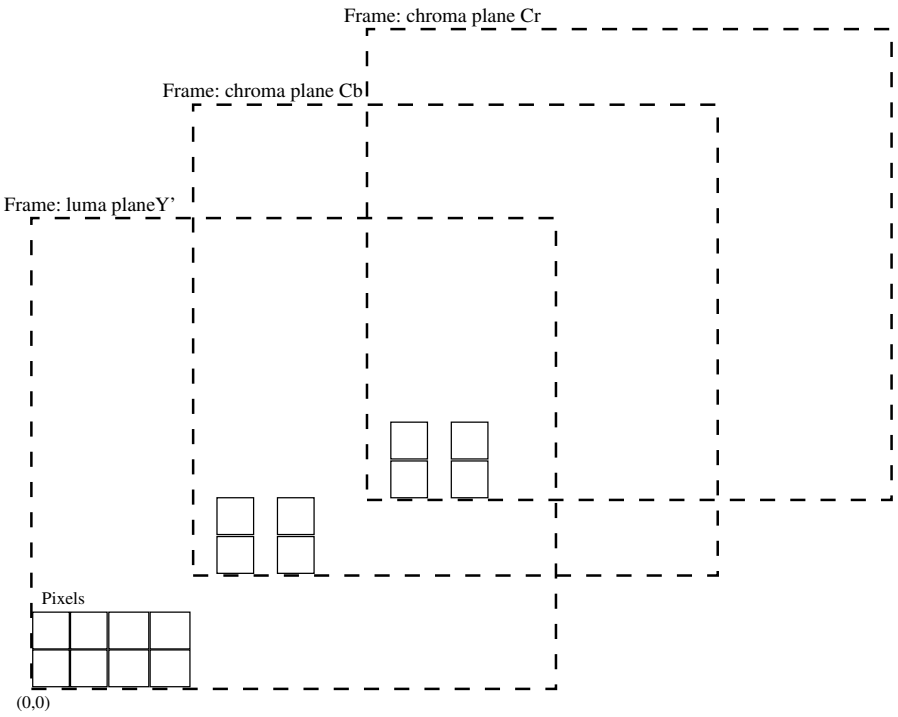


Figure 4.2: Pixels encoded 4:2:2

4.4.3 4:2:0 Subsampling

The C_b and C_r planes are stored with half the horizontal and half the vertical resolution of the Y' plane. Thus, each of these planes has half the number of horizontal blocks and half the number of vertical blocks as the luma plane, for a total of one quarter the number of blocks (see Figure 4.3). Similarly, they have half the number of horizontal super blocks and half the number of vertical super blocks, rounded up. Macro blocks are defined across color planes, and so their

number does not change, but each macro block contains within it one quarter as many chroma blocks.

The chroma samples are vertically and horizontally centered between four luma samples. Thus, each luma sample has a unique closest chroma sample. This is the same sub-sampling pattern used with JPEG, MJPEG, and MPEG-1, and was inherited from VP3. A horizontal or vertical phase shift may be required to produce signals which use different chroma sampling locations for compatibility with different systems.

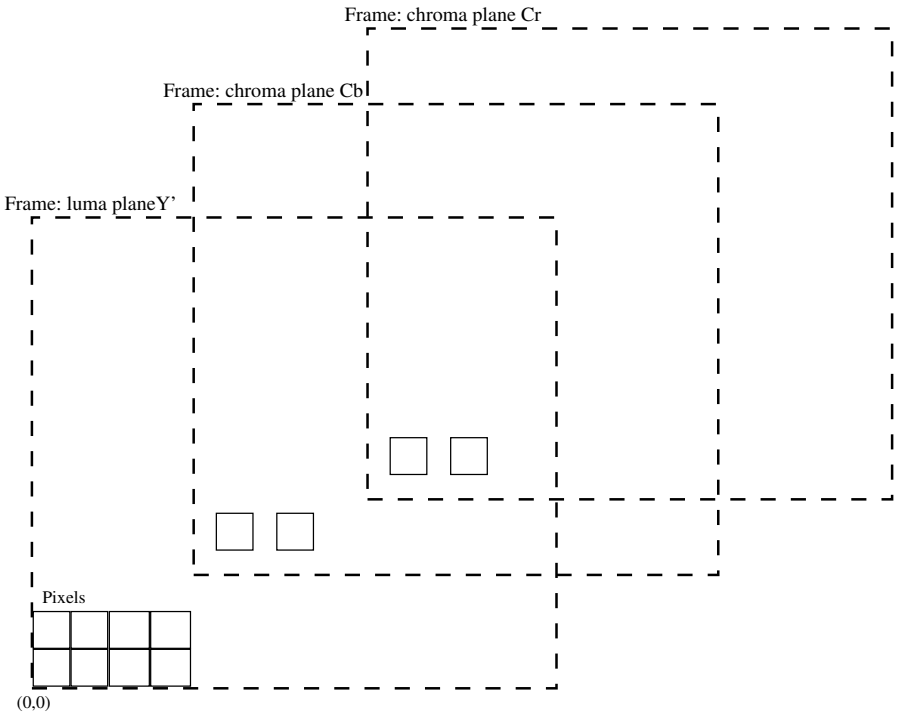


Figure 4.3: Pixels encoded 4:2:0

4.4.4 Subsampling and the Picture Region

Although the frame size must be an integral number of macro blocks, and thus both the number of pixels and the number of blocks in each direction must be even, no such requirement is made of the picture region. Thus, when using sub-sampled pixel formats, careful attention must be paid to which chroma samples correspond to which luma samples.

As mentioned above, for each pixel format, there is a unique chroma sample that is the closest to each luma sample. When cropping the chroma planes to the picture region, all the chroma samples corresponding to a luma sample in

the cropped picture region must be included. Thus, when dividing the width or height of the picture region by two to obtain the size of the subsampled chroma planes, they must be rounded up.

Furthermore, the sampling locations are defined relative to the frame, *not* the picture region. When using the 4:2:2 and 4:2:0 formats, the locations of chroma samples relative to the luma samples depends on whether or not the X offset of the picture region is odd. If the offset is even, each column of chroma samples corresponds to two columns of luma samples (see Figure 4.4 for an example). The only exception is if the width is odd, in which case the last column corresponds to only one column of luma samples (see Figure 4.5). If the offset is odd, then the first column of chroma samples corresponds to only one column of luma samples, while the remaining columns each correspond to two (see Figure 4.6). In this case, if the width is even, the last column again corresponds to only one column of luma samples (see Figure 4.7).

A similar process is followed with the rows of a picture region of odd height encoded in the 4:2:0 format. If the Y offset is even, each row of chroma samples corresponds to two rows of luma samples (see Figure 4.4), except with an odd height, where the last row corresponds to one row of chroma luna samples only (see Figure 4.5). If the offset is odd, then it is the first row of chroma samples which corresponds to only one row of luma samples, while the remaining rows each correspond to two (Figure 4.6), except with an even height, where the last row also corresponds to one (Figure 4.7).

Encoders should be aware of these differences in the subsampling when using an even or odd offset. In the typical case, with an even width and height, where one expects two rows or columns of luma samples for every row or column of chroma samples, the encoder must take care to ensure that the offsets used are both even.

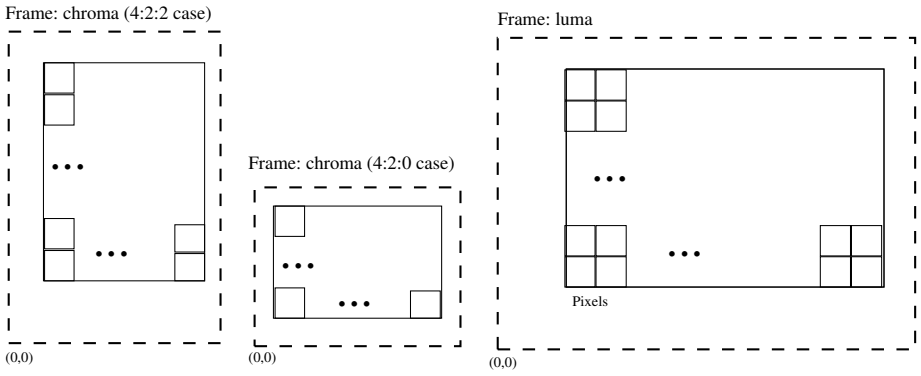


Figure 4.4: Pixel correspondence between color planes with even picture offset and even picture size

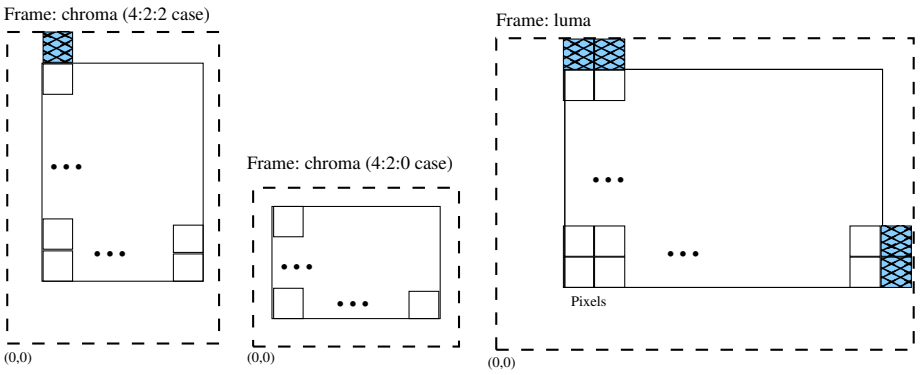


Figure 4.5: Pixel correspondence with even picture offset and odd picture size

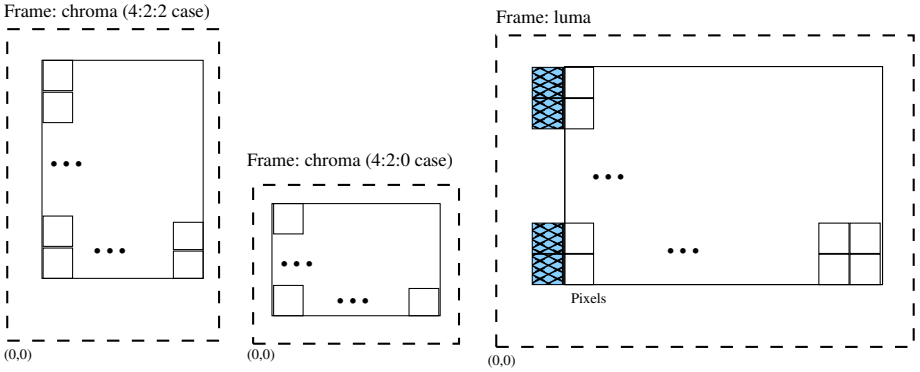


Figure 4.6: Pixel correspondence with odd picture offset and odd picture size

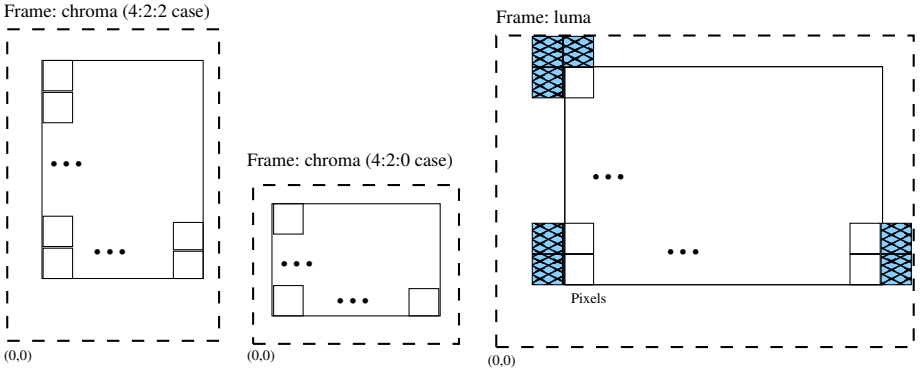


Figure 4.7: Pixel correspondence with odd picture offset and even picture size

Chapter 5

Bitpacking Convention

5.1 Overview

The Theora codec uses relatively unstructured raw packets containing binary integer fields of arbitrary width. Logically, each packet is a bitstream in which bits are written one-by-one by the encoder and then read one-by-one in the same order by the decoder. Most current binary storage arrangements group bits into a native storage unit of eight bits (octets), sixteen bits, thirty-two bits, or less commonly other fixed sizes. The Theora bitpacking convention specifies the correct mapping of the logical packet bitstream into an actual representation in fixed-width units.

5.1.1 Octets and Bytes

In most contemporary architectures, a 'byte' is synonymous with an 'octect', that is, eight bits. For purposes of the bitpacking convention, a byte implies the smallest native integer storage representation offered by a platform. Modern file systems invariably offer bytes as the fundamental atom of storage.

The most ubiquitous architectures today consider a 'byte' to be an octet. Note, however, that the Theora bitpacking convention is still well defined for any native byte size; an implementation can use the native bit-width of a given storage system. This document assumes that a byte is one octet for purposes of example only.

5.1.2 Words and Byte Order

A 'word' is an integer size that is a grouped multiple of the byte size. Most architectures consider a word to be a group of two, four, or eight bytes. Each byte in the word can be ranked by order of 'significance', e.g. the significance of the bits in each byte when storing a binary integer in the word. Several byte orderings are possible in a word. The common ones are

- Big-endian: in which the most significant byte comes first, e.g. 3-2-1-0,
- Little-endian: in which the least significant byte comes first, e.g. 0-1-2-3, and
- Mixed-endian: one of the less-common orderings that cannot be put into the above two categories, e.g. 3-1-2-0 or 0-2-1-3.

The Theora bitpacking convention specifies storage and bitstream manipulation at the byte, not word, level. Thus host word ordering is of a concern only during optimization, when writing code that operates on a word of storage at a time rather than a byte. Logically, bytes are always encoded and decoded in order from byte zero through byte n.

5.1.3 Bit Order

A byte has a well-defined 'least significant' bit (LSb), which is the only bit set when the byte is storing the two's complement integer value +1. A byte's 'most significant' bit (MSb) is at the opposite end. Bits in a byte are numbered from zero at the LSb to n for the MSb, where $n = 7$ in an octet.

5.2 Coding Bits into Bytes

The Theora codec needs to encode arbitrary bit-width integers from zero to 32 bits wide into packets. These integer fields are not aligned to the boundaries of the byte representation; the next field is read at the bit position immediately after the end of the previous field.

The decoder logically unpacks integers by first reading the MSb of a binary integer from the logical bitstream, followed by the next most significant bit, etc., until the required number of bits have been read. When unpacking the bytes into bits, the decoder begins by reading the MSb of the integer to be read from the most significant unread bit position of the source byte, followed by the next-most significant bit position of the destination integer, and so on up to the requested number of bits. Note that this differs from the Vorbis I codec, which begins decoding with the LSb of the source integer, reading it from the LSb of the source byte. When all the bits of the current source byte are read, decoding continues with the MSb of the next byte. Any unfilled bits in the last byte of the packet MUST be cleared to zero by the encoder.

5.2.1 Signedness

The binary integers decoded by the above process may be either signed or unsigned. This varies from integer to integer, and this specification indicates how each value should be interpreted as it is read. That is, depending on context, the three bit binary pattern `b111` can be taken to represent either '7' as an unsigned integer or '-1' as a signed, two's complement integer.

5.2.2 Encoding Example

The following example shows the state of an (8-bit) byte stream after several binary integers are encoded, including the location of the put pointer for the next bit to write to and the total length of the stream in bytes.

Encode the 4 bit unsigned integer value '12' (`b1100`) into an empty byte stream.

	7	6	5	4	3 ↓	2	1	0	
byte 0	**1**	**1**	**0**	**0**	0	0	0	0	←
byte 1	0	0	0	0	0	0	0	0	
byte 2	0	0	0	0	0	0	0	0	
byte 3	0	0	0	0	0	0	0	0	
⋮					⋮				
byte n	0	0	0	0	0	0	0	0	byte stream length: 1 byte

Continue by encoding the 3 bit signed integer value '-1' (`b111`).

	7	6	5	4	3	2	1	0 ↓	
byte 0	**1**	**1**	**0**	**0**	**1**	**1**	**1**	0	←
byte 1	0	0	0	0	0	0	0	0	
byte 2	0	0	0	0	0	0	0	0	
byte 3	0	0	0	0	0	0	0	0	
⋮					⋮				
byte n	0	0	0	0	0	0	0	0	byte stream length: 1 byte

Continue by encoding the 7 bit integer value '17' (`b0010001`).

	7	6	5	4	3	2	1 ↓	0	
byte 0	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	
byte 1	**0**	**1**	**0**	**0**	**0**	**1**	0	0	←
byte 2	0	0	0	0	0	0	0	0	
byte 3	0	0	0	0	0	0	0	0	
⋮					⋮				
byte n	0	0	0	0	0	0	0	0	byte stream length: 2 bytes

Continue by encoding the 13 bit integer value '6969' (`b11011 00111001`).

				↓					
	7	6	5	4	3	2	1	0	
byte 0	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	
byte 1	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	
byte 2	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	
byte 3	**0**	**0**	**1**	0	0	0	0	0	←
⋮				⋮					
byte n	0	0	0	0	0	0	0	0	byte stream length: 4 bytes

5.2.3 Decoding Example

The following example shows the state of the (8-bit) byte stream encoded in the previous example after several binary integers are decoded, including the location of the get pointer for the next bit to read.

Read a two bit unsigned integer from the example encoded above.

			↓						
	7	6	5	4	3	2	1	0	
byte 0	**1**	**1**	0	0	1	1	1	0	←
byte 1	0	1	0	0	0	1	1	1	
byte 2	0	1	1	0	0	1	1	1	
byte 3	0	0	1	0	0	0	0	0	byte stream length: 4 bytes

Value read: 3 (`b11`).

Read another two bit unsigned integer from the example encoded above.

					↓				
	7	6	5	4	3	2	1	0	
byte 0	**1**	**1**	**0**	**0**	1	1	1	0	←
byte 1	0	1	0	0	0	1	1	1	
byte 2	0	1	1	0	0	1	1	1	
byte 3	0	0	1	0	0	0	0	0	byte stream length: 4 bytes

Value read: 0 (`b00`).

Two things are worth noting here.

- Although these four bits were originally written as a single four-bit integer, reading some other combination of bit-widths from the bitstream is well defined. No artificial alignment boundaries are maintained in the bitstream.

- The first value is the integer '3' only because the context stated we were reading an unsigned integer. Had the context stated we were reading a signed integer, the returned value would have been the integer '-1'.

5.2.4 End-of-Packet Alignment

The typical use of bitpacking is to produce many independent byte-aligned packets which are embedded into a larger byte-aligned container structure, such as an Ogg transport bitstream. Externally, each bitstream encoded as a byte stream MUST begin and end on a byte boundary. Often, the encoded packet bitstream is not an integer number of bytes, and so there is unused space in the last byte of a packet.

When a Theora encoder produces packets for embedding in a byte-aligned container, unused space in the last byte of a packet is always zeroed during the encoding process. Thus, should this unused space be read, it will return binary zeroes. There is no marker pattern or stuffing bits that will allow the decoder to obtain the exact size, in bits, of the original bitstream. This knowledge is not required for decoding.

Attempting to read past the end of an encoded packet results in an 'end-of-packet' condition. Any further read operations after an 'end-of-packet' condition shall also return 'end-of-packet'. Unlike Vorbis, Theora does not use truncated packets as a normal mode of operation. Therefore if a decoder encounters the 'end-of-packet' condition during normal decoding, it may attempt to use the bits that were read to recover as much of encoded data as possible, signal a warning or error, or both.

5.2.5 Reading Zero Bit Integers

Reading a zero bit integer returns the value '0' and does not increment the stream pointer. Reading to the end of the packet, but not past the end, so that an 'end-of-packet' condition is not triggered, and then reading a zero bit integer shall succeed, returning '0', and not trigger an 'end-of-packet' condition. Reading a zero bit integer after a previous read sets the 'end-of-packet' condition shall fail, also returning 'end-of-packet'.

Chapter 6

Bitstream Headers

A Theora bitstream begins with three header packets. The header packets are, in order, the identification header, the comment header, and the setup header. All are required for decode compliance. An end-of-packet condition encountered while decoding the identification or setup header packets renders the stream undecodable. An end-of-packet condition encountered while decode the comment header is a non-fatal error condition, and MAY be ignored by a decoder.

VP3 Compatibility VP3 relies on the headers provided by its container, usually either AVI or Quicktime. As such, several parameters available in these headers are not available to VP3 streams. These are indicated as they appear in the sections below.

6.1 Common Header Decode

Input parameters: None.

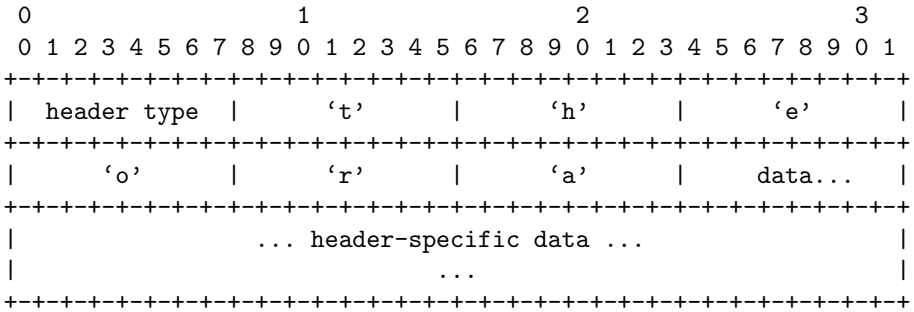


Figure 6.1: Common Header Packet Layout

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
HEADERTYPE	Integer	8	No	The type of the header being decoded.

Variables used: None.

Each header packet begins with the same header fields, which are decoded as follows:

1. Read an 8-bit unsigned integer as **HEADERTYPE**. If the most significant bit of this integer is not set, then stop. This is not a header packet.

2. Read 6 8-bit unsigned integers. If these do not have the values `0x74`, `0x68`, `0x65`, `0x6F`, `0x72`, and `0x61`, respectively, then stop. This stream is not decodable by this specification. These values correspond to the ASCII values of the characters 't', 'h', 'e', 'o', 'r', and 'a'.

Decode continues according to **HEADERTYPE**. The identification header is type `0x80`, the comment header is type `0x81`, and the setup header is type `0x82`. These packets must occur in the order: identification, comment, setup. All header packets have the most significant bit of the type field—which is the initial bit in the packet—set. This distinguishes them from video data packets in which the first bit is unset. Packets with other header types (`0x83`–`0xFF`) are reserved and MUST be ignored.

6.2 Identification Header Decode

Input parameters: None.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
VMAJ	Integer	8	No	The major version number.
VMIN	Integer	8	No	The minor version number.
VREV	Integer	8	No	The version revision number.
FMBW	Integer	16	No	The width of the frame in macro blocks.
FMBH	Integer	16	No	The height of the frame in macro blocks.

Name	Type	Size (bits)	Signed?	Description and restrictions
NSBS	Integer	32	No	The total number of super blocks in a frame.
NBS	Integer	36	No	The total number of blocks in a frame.
NMBS	Integer	32	No	The total number of macro blocks in a frame.
PICW	Integer	20	No	The width of the picture region in pixels.
PICH	Integer	20	No	The height of the picture region in pixels.
PICX	Integer	8	No	The X offset of the picture region in pixels.
PICY	Integer	8	No	The Y offset of the picture region in pixels.
FRN	Integer	32	No	The frame-rate numerator.
FRD	Integer	32	No	The frame-rate denominator.
PARN	Integer	24	No	The pixel aspect-ratio numerator.
PARD	Integer	24	No	The pixel aspect-ratio denominator.
CS	Integer	8	No	The color space.
PF	Integer	2	No	The pixel format.
NOMBR	Integer	24	No	The nominal bitrate of the stream, in bits per second.
QUAL	Integer	6	No	The quality hint.
KFGSHIFT	Integer	5	No	The amount to shift the key frame number by in the granule position.

Variables used: None.

The identification header is a short header with only a few fields used to declare the stream definitively as Theora and provide detailed information about the format of the fully decoded video data. The identification header is decoded as follows:

1. Decode the common header fields according to the procedure described in Section 6.1. If **HEADERTYPE** returned by this procedure is not `0x80`, then stop. This packet is not the identification header.

2. Read an 8-bit unsigned integer as **VMAJ**. If **VMAJ** is not 3, then stop. This stream is not decodable according to this specification.

3. Read an 8-bit unsigned integer as **VMIN**. If **VMIN** is not 2, then stop. This stream is not decodable according to this specification.

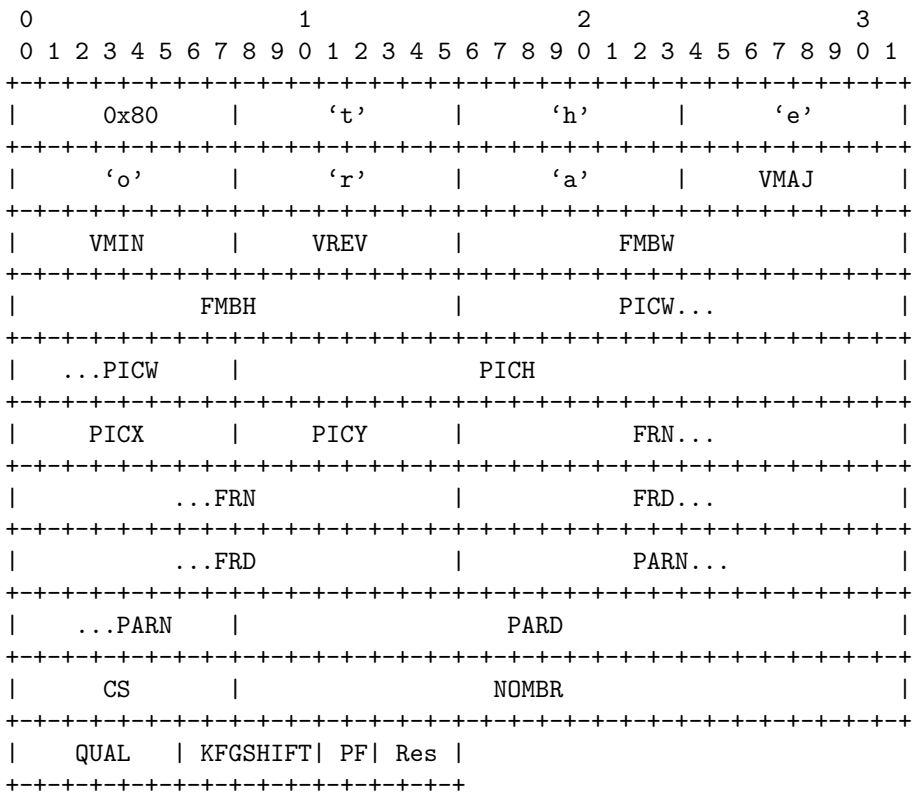


Figure 6.2: Identification Header Packet

4. Read an 8-bit unsigned integer as **VREV**. If **VREV** is greater than 1, then this stream may contain optional features or interpretational changes documented in a future version of this specification. Regardless of the value of **VREV**, the stream is decodable according to this specification.

5. Read a 16-bit unsigned integer as **FMBW**. This MUST be greater than zero. This specifies the width of the coded frame in macro blocks. The actual width of the frame in pixels is **FMBW** $*$ 16.

6. Read a 16-bit unsigned integer as **FMBH**. This MUST be greater than zero. This specifies the height of the coded frame in macro blocks. The actual height of the frame in pixels is **FMBH** $*$ 16.

7. Read a 24-bit unsigned integer as **PICW**. This MUST be no greater than (**FMBW** $*$ 16). Note that 24 bits are read, even though only 20 bits are sufficient to specify any value of the picture width. This is done to preserve octet alignment in this header, to allow for a simplified parser implementation.

8. Read a 24-bit unsigned integer as **PICH**. This MUST be no greater than (**FMBH** $*$ 16). Together with **PICW**, this specifies the size of the displayable picture region within the coded frame. See Figure 2.1. Again, 24 bits are read instead of 20.

9. Read an 8-bit unsigned integer as **PICX**. This MUST be no greater than (**FMBW** $*$ 16 $-$ **PICX**).

10. Read an 8-bit unsigned integer as **PICY**. This MUST be no greater than (**FMBH** $*$ 16 $-$ **PICY**). Together with **PICX**, this specifies the location of the lower-left corner of the displayable picture region. See Figure 2.1.

11. Read a 32-bit unsigned integer as **FRN**. This MUST be greater than zero.

12. Read a 32-bit unsigned integer as **FRD**. This MUST be greater than zero. Theora is a fixed-frame rate video codec. Frames are sampled at the constant rate of $\frac{\mathbf{FRN}}{\mathbf{FRD}}$ frames per second. The presentation time of the first frame is at zero seconds. No mechanism is provided to specify a non-zero offset for the initial frame.

13. Read a 24-bit unsigned integer as **PARN**.

14. Read a 24-bit unsigned integer as **PARD**. Together with **PARN**, these specify the aspect ratio of the pixels within a frame, defined as the ratio of the physical width of a pixel to its physical height. This is given by the ratio **PARN** : **PARD**. If either of these fields are zero, this indicates that pixel aspect ratio information was not available to the encoder. In this case it MAY be specified by the application via an external means, or a default value of 1 : 1 MAY be used.

15. Read an 8-bit unsigned integer as **CS**. This is a value from an enumerated list of the available color spaces, given in Table 6.3. The 'Undefined' value indicates that color space information was not available to the encoder. It MAY be specified by the application via an external means. If a reserved value is given, a decoder MAY refuse to decode the stream.

Value	Color Space
0	Undefined.
1	Rec. 470M (see Section 4.3.1).
2	Rec. 470BG (see Section 4.3.2).
3	Reserved.
⋮	
255	

Table 6.3: Enumerated List of Color Spaces

16. Read a 24-bit unsigned integer as **NOMBR**. The **NOMBR** field is used only as a hint. For pure VBR streams, this value may be considerably off. The field MAY be set to zero to indicate that the encoder did not care to speculate.

17. Read a 6-bit unsigned integer as **QUAL**. This value is used to provide a hint as to the relative quality of the stream when compared to others produced by the same encoder. Larger values indicate higher quality. This can be used, for example, to select among several streams containing the same material encoded with different settings.

18. Read a 5-bit unsigned integer as **KFGSHIFT**. The **KFGSHIFT** is used to partition the granule position associated with each packet into two different parts. The frame number of the last key frame, starting from zero, is stored in the upper 64−**KFGSHIFT** bits, while the lower **KFGSHIFT** bits contain the number of frames since the last keyframe. Complete details on the granule position mapping are specified in Section REF.

19. Read a 2-bit unsigned integer as **PF**. The **PF** field contains a value from an enumerated list of the available pixel formats, given in Table 6.4. If the reserved value 1 is given, stop. This stream is not decodable according to this specification.

20. Read a 3-bit unsigned integer. These bits are reserved. If this value is not zero, then stop. This stream is not decodable according to this specification.

21. Assign **NSBS** a value according to **PF**, as given by Table 6.5.

22. Assign **NBS** a value according to **PF**, as given by Table 6.6.

Value	Pixel Format
0	4:2:0 (see Section 4.4.3).
1	Reserved.
2	4:2:2 (see Section 4.4.2).
3	4:4:4 (see Section 4.4.1).

Table 6.4: Enumerated List of Pixel Formats

PF	**NSBS**
0	$((\mathbf{FMBW}+1)//2)*((\mathbf{FMBH}+1)//2)$ $+2*((\mathbf{FMBW}+3)//4)*((\mathbf{FMBH}+3)//4)$
2	$((\mathbf{FMBW}+1)//2)*((\mathbf{FMBH}+1)//2)$ $+2*((\mathbf{FMBW}+3)//4)*((\mathbf{FMBH}+1)//2)$
3	$3*((\mathbf{FMBW}+1)//2)*((\mathbf{FMBH}+1)//2)$

Table 6.5: Number of Super Blocks for each Pixel Format

23. Assign **NMBS** the value ($\mathbf{FMBW}*\mathbf{FMBH}$).

VP3 Compatibility VP3 does not correctly handle frame sizes that are not a multiple of 16. Thus, **PICW** and **PICH** should be set to the frame width and height in pixels, respectively, and **PICX** and **PICY** should be set to zero. VP3 headers do not specify a color space. VP3 only supports the 4:2:0 pixel format.

6.3 Comment Header

The Theora comment header is the second of three header packets that begin a Theora stream. It is meant for short text comments, not aribtrary metadata; arbitrary metadata belongs in a separate logical stream that provides greater structure and machine parseability.

PF	**NBS**
0	$6*\mathbf{FMBW}*\mathbf{FMBH}$
2	$8*\mathbf{FMBW}*\mathbf{FMBH}$
3	$12*\mathbf{FMBW}*\mathbf{FMBH}$

Table 6.6: Number of Blocks for each Pixel Format

4 byte length	UTF-8 encoded string ...

Figure 6.3: Length encoded string layout

The comment field is meant to be used much like someone jotting a quick note on the label of a video. It should be a little information to remember the disc or tape by and explain it to others; a short, to-the-point text note that can be more than a couple words, but isn't going to be more than a short paragraph. The essentials, in other words, whatever they turn out to be, e.g.:

The comment header is stored as a logical list of eight-bit clean vectors; the number of vectors is bounded at $2^{32}-1$ and the length of each vector is limited to $2^{32}-1$ bytes. The vector length is encoded; the vector contents themselves are not null terminated. In addition to the vector list, there is a single vector for a vendor name, also eight-bit clean with a length encoded in 32 bits.

6.3.1 Comment Length Decode

Input parameters: None.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
LEN	Integer	32	No	A single 32-bit length value.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
LEN0	Integer	8	No	The first octet of the string length.
LEN1	Integer	8	No	The second octet of the string length.
LEN2	Integer	8	No	The third octet of the string length.
LEN3	Integer	8	No	The fourth octet of the string length.

A single comment vector is decoded as follows:

1. Read an 8-bit unsigned integer as LEN0.
2. Read an 8-bit unsigned integer as LEN1.
3. Read an 8-bit unsigned integer as LEN2.
4. Read an 8-bit unsigned integer as LEN3.
5. Assign **LEN** the value (LEN0+(LEN1 << 8)+(LEN2 << 16)+(LEN3 << 24)). This construction is used so that on platforms with 8-bit bytes, the

vendor string
number of comments
comment string
comment string
...

Figure 6.4: Comment Header Layout

memory organization of the comment header is identical with that of Vorbis I, allowing for common parsing code despite the different bit packing conventions.

6.3.2 Comment Header Decode

Input parameters: None.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
VENDOR	String			The vendor string.
NCOMMENTS	Integer	32	No	The number of user comments.
COMMENTS	String Array			A list of **NCOMMENTS** user comment values.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
ci	Integer	32	No	The index of the current user comment.

The complete comment header is decoded as follows:

1. Decode the common header fields according to the procedure described in Section 6.1. If **HEADERTYPE** returned by this procedure is not `0x81`, then stop. This packet is not the comment header.
2. Decode the length of the vendor string using the procedure given in Section 6.3.1 into **LEN**.
3. Read **LEN** 8-bit unsigned integers.
4. Set the string **VENDOR** to the contents of these octets.
5. Decode the number of user comments using the procedure given in Section 6.3.1 into **LEN**.
6. Assign **NCOMMENTS** the value stored in **LEN**.
7. For each consecutive value of *ci* from 0 to (**NCOMMENTS** − 1), inclusive:

(a) Decode the length of the current user comment using the procedure given in Section 6.3.1 into **LEN**.

(b) Read **LEN** 8-bit unsigned integers.

(c) Set the string **COMMENTS**[*ci*] to the contents of these octets.

The comment header comprises the entirety of the second header packet. Unlike the first header packet, it is not generally the only packet on the second page and may span multiple pages. The length of the comment header packet is (practically) unbounded. The comment header packet is not optional; it must be present in the stream even if it is logically empty.

6.3.3 User Comment Format

The user comment vectors are structured similarly to a UNIX environment variable. That is, comment fields consist of a field name and a corresponding value and look like:

COMMENTS[0] = "TITLE=the look of Theora"
COMMENTS[1] = "DIRECTOR=me"

The field name is case-insensitive and MUST consist of ASCII characters `0x20` through `0x7D`, `0x3D` ('=') excluded. ASCII `0x41` through `0x5A` inclusive (characters 'A'–'Z') are to be considered equivalent to ASCII `0x61` through `0x7A` inclusive (characters 'a'–'z'). An entirely empty field name—one that is zero characters long—is not disallowed.

The field name is immediately followed by ASCII `0x3D` ('='); this equals sign is used to terminate the field name.

The data immediately after `0x3D` until the end of the vector is the eight-bit clean value of the field contents encoded as a UTF-8 string [Yer96].

Field names MUST NOT be 'internationalized'; this is a concession to simplicity, not an attempt to exclude the majority of the world that doesn't speak English. Applications MAY wish to present internationalized versions of the standard field names listed below to the user, but they are not to be stored in the bitstream. Field *contents*, however, use the UTF-8 character encoding to allow easy representation of any language.

Individual 'vendors' MAY use non-standard field names within reason. The proper use of comment fields as human-readable notes has already been explained. Abuse will be discouraged.

There is no vendor-specific prefix to 'non-standard' field names. Vendors SHOULD make some effort to avoid arbitrarily polluting the common namespace. Xiph.org and other bodies will generally collect and rationalize the more useful tags to help with standardization.

Field names are not restricted to occur only once within a comment header.

Field Names Below is a proposed, minimal list of standard field names with a description of their intended use. No field names are mandatory; a comment header may contain one or more, all, or none of the names in this list.

TITLE: Video name.

ARTIST: Filmmaker or other creator name.

VERSION: Subtitle, remix info, or other text distinguishing versions of a video.

DATE: Date associated with the video. Implementations SHOULD attempt to parse this field as an ISO 8601 date for machine interpretation and conversion.

LOCATION: Location associated with the video. This is usually the filming location for non-fiction works.

COPYRIGHT: Copyright statement.

LICENSE: Copyright and other licensing information. Implementations wishing to do automatic parsing of e.g of distribution terms SHOULD look here for a URL uniquely defining the license. If no instance of this field is present, or if no instance contains a parseable URL, and implementation MAY look in the COPYRIGHT field for such a URL.

ORGANIZATION: Studio name, Publisher, or other organization involved in the creation of the video.

DIRECTOR: Director or Filmmaker credit, similar to ARTIST.

PRODUCER: Producer credit for the video.

COMPOSER: Music credit for the video.

ACTOR: Acting credit for the video.

TAG: subject or category tag, keyword, or other content classification labels. The value of each instance of this field SHOULD be treated as a single label, with multiple instances of the field for multiple tags. The value of a single field SHOULD NOT be parsed into multiple tags based on some internal delimeter.

DESCRIPTION: General description, summary, or blurb.

6.4 Setup Header

The Theora setup header contains the limit values used to drive the loop filter, the base matrices and scale values used to build the dequantization tables, and the Huffman tables used to unpack the DCT tokens. Because the contents of this header are specific to Theora, no concessions have been made to keep the fields octet-aligned for easy parsing.

6.4.1 Loop Filter Limit Table Decode

Input parameters: None.

common header block
loop filter table resolution
loop filter table
scale table resolution
AC scale table
DC scale table
number of base matricies
base quatization matricies
...
quant range interpolation table
DCT token Huffman tables

Figure 6.5: Setup Header structure

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
LFLIMS	Integer array	7	No	A 64-element array of loop filter limit values.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
qi	Integer	6	No	The quantization index.
NBITS	Integer	3	No	The size of values being read in the current table.

This procedure decodes the table of loop filter limit values used to drive the loop filter, which is described in Section 6.4.1. It is decoded as follows:

1. Read a 3-bit unsigned integer as NBITS.
2. For each consecutive value of qi from 0 to 63, inclusive:
   (a) Read an NBITS-bit unsigned integer as **LFLIMS**[qi].

VP3 Compatibility The loop filter limit values are hardcoded in VP3. The values used are given in Appendix B.2.

6.4.2 Quantization Parameters Decode

Input parameters: None.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions

Name	Type	Size (bits)	Signed?	Description and restrictions
NBMS	Integer	10	No	The number of base matrices.
BMS	2D Integer array	8	No	A **NBMS** $\times 64$ array containing the base matrices.
NQRS	2D Integer array	6	No	A 2×3 array containing the number of quant ranges for a given *qti* and *pli*, respectively. This is at most 63.
QRSIZES	3D Integer array	6	No	A $2 \times 3 \times 63$ array of the sizes of each quant range for a given *qti* and *pli*, respectively. Only the first **NQRS**[*qti*][*pli*] values are used.
QRBMIS	3D Integer array	9	No	A $2 \times 3 \times 64$ array of the *bmi*'s used for each quant range for a given *qti* and *pli*, respectively. Only the first (**NQRS**[*qti*][*pli*] + 1) values are used.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
qti	Integer	1	No	A quantization type index. See Table 3.1.
qtj	Integer	1	No	A quantization type index.
pli	Integer	2	No	A color plane index. See Table 2.1.
plj	Integer	2	No	A color plane index.
qi	Integer	6	No	The quantization index.
ci	Integer	6	No	The DCT coefficient index.
bmi	Integer	9	No	The base matrix index.
qri	Integer	6	No	The quant range index.
NBITS	Integer	5	No	The size of fields to read.
NEWQR	Integer	1	No	Flag that indicates a new set of quant ranges will be defined.
RPQR	Integer	1	No	Flag that indicates the quant ranges to copy will come from the same color plane.

The AC scale and DC scale values are defined in two simple tables with 64

values each, one for each qi value. The same scale values are used for every quantization type and color plane.

The base matrices for all quantization types and color planes are stored in a single table. These are then referenced by index in several sets of *quant ranges*. The purpose of the quant ranges is to specify which base matrices are used for which qi values.

A set of quant ranges is defined for each quantization type and color plane. To save space in the header, bit flags allow a set of quant ranges to be copied from a previously defined set instead of being specified explicitly. Every set except the first one can be copied from the immediately preceding set. Similarly, if the quantization type is not 0, the set can be copied from the set defined for the same color plane for the preceding quantization type. This formulation allows compact representation of, for example, the same set of quant ranges in both chroma channels, as is done in the original VP3, or the same set of quant ranges in INTRA and INTER modes.

Each quant range is defined by a size and two base matrix indices, one for each end of the range. The base matrix for the end of one range is used as the start of the next range, so that for n ranges, $n + 1$ base matrices are specified. The base matrices for the qi values between the two endpoints of the range are generated by linear interpolation.

The location of the endpoints of each range is encoded by their size. The qi value for the left end-point is the sum of the sizes of all preceding ranges, and the qi value for the right end-point adds the size of the current range. Thus the sum of the sizes of all the ranges MUST be 63, so that the last range falls on the last possible qi value.

The complete set of quantization parameters are decoded as follows:

1. Read a 4-bit unsigned integer. Assign NBITS the value read, plus one.
2. For each consecutive value of qi from 0 to 63, inclusive:
   (a) Read an NBITS-bit unsigned integer as **ACSCALE**[qi].
3. Read a 4-bit unsigned integer. Assign NBITS the value read, plus one.
4. For each consecutive value of qi from 0 to 63, inclusive:
   (a) Read an NBITS-bit unsigned integer as **DCSCALE**[qi].
5. Read a 9-bit unsigned integer. Assign **NBMS** the value decoded, plus one. **NBMS** MUST be no greater than 384.
6. For each consecutive value of bmi from 0 to (**NBMS** $- 1$), inclusive:
   (a) For each consecutive value of ci from 0 to 63, inclusive:
      i. Read an 8-bit unsigned integer as **BMS**[bmi][ci].
7. For each consecutive value of qti from 0 to 1, inclusive:

(a) For each consecutive value of *pli* from 0 to 2, inclusive:

i. If $qti > 0$ or $pli > 0$, read a 1-bit unsigned integer as NEWQR.

ii. Else, assign NEWQR the value one.

iii. If NEWQR is zero, then we are copying a previously defined set of quant ranges. In that case:

A. If $qti > 0$, read a 1-bit unsigned integer as RPQR.

B. Else, assign RPQR the value zero.

C. If RPQR is one, assign *qtj* the value $(qti - 1)$ and assign *plj* the value *pli*. This selects the set of quant ranges defined for the same color plane as this one, but for the previous quantization type.

D. Else assign *qtj* the value $(3 * qti + pli - 1)//3$ and assign *plj* the value $(pli + 2)\%3$. This selects the most recent set of quant ranges defined.

E. Assign $\mathbf{NQRS}[qti][pli]$ the value $\mathbf{NQRS}[qtj][plj]$.

F. Assign $\mathbf{QRSIZES}[qti][pli]$ the values in $\mathbf{QRSIZES}[qtj][plj]$.

G. Assign $\mathbf{QRBMIS}[qti][pli]$ the values in $\mathbf{QRBMIS}[qtj][plj]$.

iv. Else, NEWQR is one, which indicates that we are defining a new set of quant ranges. In that case:

A. Assign *qri* the value zero.

B. Assign *qi* the value zero.

C. Read an $\text{ilog}(\mathbf{NBMS} - 1)$-bit unsigned integer as $\mathbf{QRBMIS}[qti][pli][qri]$. If this is greater than or equal to **NBMS**, stop. The stream is undecodable.

D. Read an $\text{ilog}(62 - qi)$-bit unsigned integer. Assign $\mathbf{QRSIZES}[qti][pli][qri]$ the value read, plus one.

E. Assign *qi* the value $qi + \mathbf{QRSIZES}[qti][pli][qri]$.

F. Assign *qri* the value $qri + 1$.

G. Read an $\text{ilog}(\mathbf{NBMS} - 1)$-bit unsigned integer as $\mathbf{QRBMIS}[qti][pli][qri]$.

H. If *qi* is less than 63, go back to step 7(a)ivD.

I. If *qi* is greater than 63, stop. The stream is undecodable.

J. Assign $\mathbf{NQRS}[qti][pli]$ the value *qri*.

VP3 Compatibility The quantization parameters are hardcoded in VP3. The values used are given in Appendix B.3.

6.4.3 Computing a Quantization Matrix

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
ACSCALE	Integer array	16	No	A 64-element array of scale values for AC coefficients for each *qi* value.
DCSCALE	Integer array	16	No	A 64-element array of scale values for the DC coefficient for each *qi* value.
BMS	2D Integer array	8	No	A **NBMS** $\times$ 64 array containing the base matrices.
NQRS	2D Integer array	6	No	A 2×3 array containing the number of quant ranges for a given *qti* and *pli*, respectively. This is at most 63.
QRSIZES	3D Integer array	6	No	A $2 \times 3 \times 63$ array of the sizes of each quant range for a given *qti* and *pli*, respectively. Only the first **NQRS**[*qti*][*pli*] values are used.
QRBMIS	3D Integer array	9	No	A $2 \times 3 \times 64$ array of the *bmi*'s used for each quant range for a given *qti* and *pli*, respectively. Only the first (**NQRS**[*qti*][*pli*] + 1) values are used.
qti	Integer	1	No	A quantization type index. See Table 3.1.
pli	Integer	2	No	A color plane index. See Table 2.1.
qi	Integer	6	No	The quantization index.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
QMAT	Integer array	16	No	A 64-element array of quantization values for each DCT coefficient in natural order.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
ci	Integer	6	No	The DCT coefficient index.
bmi	Integer	9	No	The base matrix index.
bmj	Integer	9	No	The base matrix index.
qri	Integer	6	No	The quant range index.
QISTART	Integer	6	No	The left end-point of the *qi* range.
QIEND	Integer	6	No	The right end-point of the *qi* range.
BM	Integer array	8	No	A 64-element array containing the interpolated base matrix.
QMIN	Integer	16	No	The minimum quantization value allowed for the current coefficient.
QSCALE	Integer	16	No	The current scale value.

The following procedure can be used to generate a single quantization matrix for a given quantization type, color plane, and *qi* value, given the quantization parameters decoded in Section 6.4.2.

Note that the product of the scale value and the base matrix value is in units of 100ths of a pixel value, and thus is divided by 100 to return it to units of a single pixel value. This value is then scaled by four, to match the scaling of the DCT output, which is also a factor of four larger than the orthonormal version of the transform.

1. Assign *qri* the index of a quant range such that

$$\sum_{qrj=0}^{qri-1} \boldsymbol{qi} \geq \mathbf{QRSIZES}[\boldsymbol{qti}][\boldsymbol{pli}][qrj],$$

   and

$$\sum_{qrj=0}^{qri} \boldsymbol{qi} \leq \mathbf{QRSIZES}[\boldsymbol{qti}][\boldsymbol{pli}][qrj],$$

   where summation from 0 to −1 is defined to be zero. If there is more than one such value of *qri*, i.e., if ***qi*** lies on the boundary between two quant ranges, then the output will be the same regardless of which one is chosen.

2. Assign QISTART the value

$$\sum_{qrj=0}^{qri-1} \mathbf{QRSIZES}[\boldsymbol{qti}][\boldsymbol{pli}][qrj].$$

3. Assign QIEND the value

$$\sum_{qrj=0}^{qri} \mathbf{QRSIZES}[\boldsymbol{qti}][\boldsymbol{pli}][qrj].$$

4. Assign bmi the value $\mathbf{QRBMIS}[\boldsymbol{qti}][\boldsymbol{pli}][qri]$.
5. Assign bmj the value $\mathbf{QRBMIS}[\boldsymbol{qti}][\boldsymbol{pli}][qri + 1]$.
6. For each consecutive value of ci from 0 to 63, inclusive:
   (a) Assign BM$[ci]$ the value

$$\begin{aligned}&(2 * (\text{QIEND} - \boldsymbol{qi}) * \mathbf{BMS}[bmi][ci]\\ &+ 2 * (\boldsymbol{qi} - \text{QISTART}) * \mathbf{BMS}[bmj][ci]\\ &+ \mathbf{QRSIZES}[\boldsymbol{qti}][\boldsymbol{pli}][qri])//(2 * \mathbf{QRSIZES}[\boldsymbol{qti}][\boldsymbol{pli}][qri])\end{aligned}$$

   (b) Assign QMIN the value given by Table 6.18 according to ***qti*** and ci.

Coefficient	***qti***	QMIN
$ci = 0$	0 (Intra)	16
$ci > 0$	0 (Intra)	8
$ci = 0$	1 (Inter)	32
$ci > 0$	1 (Inter)	16

Table 6.18: Minimum Quantization Values

   (c) If ci equals zero, assign QSCALE the value $\mathbf{DCSCALE}[\boldsymbol{qi}]$.
   (d) Else, assign QSCALE the value $\mathbf{ACSCALE}[\boldsymbol{qi}]$.
   (e) Assign $\mathbf{QMAT}[ci]$ the value

$$\max(\text{QMIN}, \min((\text{QSCALE} * \text{BM}[ci]//100) * 4, 4096)).$$

6.4.4 DCT Token Huffman Tables

Input parameters: None.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
HTS	Huffman table array			An 80-element array of Huffman tables with up to 32 entries each.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
HBITS	Bit string	32	No	A string of up to 32 bits.
TOKEN	Integer	5	No	A single DCT token value.
ISLEAF	Integer	1	No	Flag that indicates if the current node of the tree being decoded is a leaf node.

The Huffman tables used to decode DCT tokens are stored in the setup header in the form of a binary tree. This enforces the requirements that the code be full—so that any sequence of bits will produce a valid sequence of tokens—and that the code be prefix-free so that there is no ambiguity when decoding.

One more restriction is placed on the tables that is not explicitly enforced by the bitstream syntax, but nevertheless must be obeyed by compliant encoders. There must be no more than 32 entries in a single table. Note that this restriction along with the fullness requirement limit the maximum size of a single Huffman code to 32 bits. It is probably a good idea to enforce this latter consequence explicitly when implementing the decoding procedure as a recursive algorithm, so as to prevent a possible stack overflow given an invalid bitstream.

Although there are 32 different DCT tokens, and thus a normal table will have exactly 32 entries, this is not explicitly required. It is allowable to use a Huffman code that omits some—but not all—of the possible token values. It is also allowable, if not particularly useful, to specify multiple codes for the same token value in a single table. Note also that token values may appear in the tree in any order. In particular, it is not safe to assume that token value zero (which ends a single block), has a Huffman code of all zeros.

The tree is decoded as follows:

1. For each consecutive value of *hti* from 0 to 79, inclusive:
   (a) Set HBITS to the empty string.
   (b) If HBITS is longer than 32 bits in length, stop. The stream is undecodable.
   (c) Read a 1-bit unsigned integer as ISLEAF.
   (d) If ISLEAF is one:
      i. If the number of entries in table **HTS**[*hti*] is already 32, stop. The stream is undecodable.
      ii. Read a 5-bit unsigned integer as TOKEN.
      iii. Add the pair (HBITS, TOKEN) to Huffman table **HTS**[*hti*].
   (e) Otherwise:
      i. Add a '0' to the end of HBITS.
      ii. Decode the '0' sub-tree using this procedure, starting from step 1b.

iii. Remove the '0' from the end of HBITS and add a '1' to the end of HBITS.
iv. Decode the '1' sub-tree using this procedure, starting from step 1b.
v. Remove the '1' from the end of HBITS.

VP3 Compatibility The DCT token Huffman tables are hardcoded in VP3. The values used are given in Appendix B.4.

6.4.5 Setup Header Decode

Input parameters: None.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
LFLIMS	Integer array	7	No	A 64-element array of loop filter limit values.
ACSCALE	Integer array	16	No	A 64-element array of scale values for AC coefficients for each *qi* value.
DCSCALE	Integer array	16	No	A 64-element array of scale values for the DC coefficient for each *qi* value.
NBMS	Integer	10	No	The number of base matrices.
BMS	2D Integer array	8	No	A **NBMS** $\times 64$ array containing the base matrices.
NQRS	2D Integer array	6	No	A 2×3 array containing the number of quant ranges for a given *qti* and *pli*, respectively. This is at most 63.
QRSIZES	3D Integer array	6	No	A $2 \times 3 \times 63$ array of the sizes of each quant range for a given *qti* and *pli*, respectively. Only the first **NQRS**[*qti*][*pli*] values will be used.
QRBMIS	3D Integer array	9	No	A $2 \times 3 \times 64$ array of the *bmi*'s used for each quant range for a given *qti* and *pli*, respectively. Only the first (**NQRS**[*qti*][*pli*] + 1) values will be used.
HTS	Huffman table array			An 80-element array of Huffman tables with up to 32 entries each.

Name	Type	Size (bits)	Signed?	Description and restrictions

Variables used: None.

The complete setup header is decoded as follows:

1. Decode the common header fields according to the procedure described in Section 6.1. If **HEADERTYPE** returned by this procedure is not `0x82`, then stop. This packet is not the setup header.

2. Decode the loop filter limit value table using the procedure given in Section 6.4.1 into **LFLIMS**.

3. Decode the quantization parameters using the procedure given in Section 6.4.2. The results are stored in **ACSCALE**, **DCSCALE**, **NBMS**, **BMS**, **NQRS**, **QRSIZES**, and **QRBMIS**.

4. Decode the DCT token Huffman tables using the procedure given in Section 6.4.4 into **HTS**.

Chapter 7

Frame Decode

This section describes the complete procedure necessary to decode a single frame. This begins with the frame header, followed by coded block flags, macro block modes, motion vectors, block-level *qi* values, and finally the DCT residual tokens, which are used to reconstruct the frame.

7.1 Frame Header Decode

Input parameters: None.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
FTYPE	Integer	1	No	The frame type.
NQIS	Integer	2	No	The number of *qi* values.
QIS	Integer array	6	No	An **NQIS**-element array of *qi* values.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
MOREQIS	Integer	1	No	A flag indicating there are more *qi* values to be decoded.

The frame header selects which type of frame is being decoded, intra or inter,

and contains the list of *qi* values that will be used in this frame. The first *qi* value will be used for *all* DC coefficients in all blocks. This is done to ensure that DC prediction, which is done in the quantized domain, works as expected. The AC coefficients, however, can be dequantized using any *qi* value on the list, selected on a block-by-block basis.

1. Read a 1-bit unsigned integer. If the value read is not zero, stop. This is not a data packet.
2. Read a 1-bit unsigned integer as **FTYPE**. This is the type of frame being decoded, as given in Table 7.3. If this is the first frame being decoded, this MUST be zero.

FTYPE	Frame Type
0	Intra frame
1	Inter frame

Table 7.3: Frame Type Values

3. Read in a 6-bit unsigned integer as **QIS**[0].
4. Read a 1-bit unsigned integer as MOREQIS.
5. If MOREQIS is zero, set **NQIS** to 1.
6. Otherwise:
   (a) Read in a 6-bit unsigned integer as **QIS**[1].
   (b) Read a 1-bit unsigned integer as MOREQIS.
   (c) If MOREQIS is zero, set **NQIS** to 2.
   (d) Otherwise:
      i. Read in a 6-bit unsigned integer as **QIS**[2].
      ii. Set **NQIS** to 3.
7. If **FTYPE** is 0, read a 3-bit unsigned integer. These bits are reserved. If this value is not zero, stop. This frame is not decodable according to this specification.

VP3 Compatibility The precise format of the frame header is substantially different in Theora than in VP3. The original VP3 format includes a larger number of unused, reserved bits that are required to be zero. The original VP3 frame header also can contain only a single *qi* value, because VP3 does not support block-level *qi* values and uses the same *qi* value for all the coefficients in a frame.

7.2 Run-Length Encoded Bit Strings

Two variations of run-length encoding are used to store sequences of bits for the block coded flags and the block-level qi values. The procedures to decode these bit sequences are specified in the following two sections.

7.2.1 Long-Run Bit String Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
NBITS	Integer	36	No	The number of bits to decode.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
BITS	Bit string			The decoded bits.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
LEN	Integer	36	No	The number of bits decoded so far.
BIT	Integer	1	No	The value associated with the current run.
RLEN	Integer	13	No	The length of the current run.
RBITS	Integer	4	No	The number of extra bits needed to decode the run length.
RSTART	Integer	6	No	The start of the possible run-length values for a given Huffman code.
ROFFS	Integer	12	No	The offset from RSTART of the run-length.

There is no practical limit to the number of consecutive 0's and 1's that can be decoded with this procedure. In reality, the run length is limited by the number of blocks in a single frame, because more will never be requested. A separate procedure described in Section 7.2.2 is used when there is a known limit on the maximum size of the runs.

For the first run, a single bit value is read, and then a Huffman-coded representation of a run length is decoded, and that many copies of the bit value are appended to the bit string. For each consecutive run, the value of the bit is toggled instead of being read from the bitstream.

The only exception is if the length of the previous run was 4129, the maximum possible length encodable by the Huffman-coded representation. In this case another bit value is read from the stream, to allow for consecutive runs of 0's or 1's longer than this maximum.

Note that in both cases—for the first run and after a run of length 4129—if no more bits are needed, then no bit value is read.

The complete decoding procedure is as follows:

1. Assign LEN the value 0.
2. Assign **BITS** the empty string.
3. If LEN equals **NBITS**, return the completely decoded string **BITS**.
4. Read a 1-bit unsigned integer as BIT.
5. Read a bit at a time until one of the Huffman codes given in Table 7.7 is recognized.

Huffman Code	RSTART	RBITS	Run Lengths
`b0`	1	0	1
`b10`	2	1	2 . . . 3
`b110`	4	1	4 . . . 5
`b1110`	6	2	6 . . . 9
`b11110`	10	3	10 . . . 17
`b111110`	18	4	18 . . . 33
`b111111`	34	12	34 . . . 4129

Table 7.7: Huffman Codes for Long Run Lengths

6. Assign RSTART and RBITS the values given in Table 7.7 according to the Huffman code read.
7. Read an RBITS-bit unsigned integer as ROFFS.
8. Assign RLEN the value (RSTART + ROFFS).
9. Append RLEN copies of BIT to **BITS**.
10. Add RLEN to the value LEN. LEN MUST be less than or equal to **NBITS**.
11. If LEN equals **NBITS**, return the completely decoded string **BITS**.

12. If RLEN equals 4129, read a 1-bit unsigned integer as BIT.
13. Otherwise, assign BIT the value $(1 - \text{BIT})$.
14. Continue decoding runs from step 5.

VP3 Compatibility VP3 does not read a new bit value after decoding a run length of 4129. This limits the maximum number of consecutive 0's or 1's to 4129 in VP3-compatible streams. For reasonable video sizes of 1920×1080 or less in 4:2:0 format—the only pixel format VP3 supports—this does not pose any problems because runs longer than 4129 are not needed.

7.2.2 Short-Run Bit String Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
NBITS	Integer	36	No	The number of bits to decode.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
BITS	Bit string			The decoded bits.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
LEN	Integer	36	No	The number of bits decoded so far.
BIT	Integer	1	No	The value associated with the current run.
RLEN	Integer	13	No	The length of the current run.
RBITS	Integer	4	No	The number of extra bits needed to decode the run length.
RSTART	Integer	6	No	The start of the possible run-length values for a given Huffman code.
ROFFS	Integer	12	No	The offset from RSTART of the run-length.

This procedure is similar to the procedure outlined in Section 7.2.1, except that the maximum number of consecutive 0's or 1's is limited to 30. This is the maximum run length needed when encoding a bit for each of the 16 blocks in a super block when it is known that not all the bits in a super block are the same.

The complete decoding procedure is as follows:

1. Assign LEN the value 0.
2. Assign **BITS** the empty string.
3. If LEN equals **NBITS**, return the completely decoded string **BITS**.
4. Read a 1-bit unsigned integer as BIT.
5. Read a bit at a time until one of the Huffman codes given in Table 7.11 is recognized.

Huffman Code	RSTART	RBITS	Run Lengths
`b0`	1	1	1 . . . 2
`b10`	3	1	3 . . . 4
`b110`	5	1	5 . . . 6
`b1110`	7	2	7 . . . 10
`b11110`	11	2	11 . . . 14
`b11111`	15	4	15 . . . 30

Table 7.11: Huffman Codes for Short Run Lengths

6. Assign RSTART and RBITS the values given in Table 7.11 according to the Huffman code read.
7. Read an RBITS-bit unsigned integer as ROFFS.
8. Assign RLEN the value (RSTART + ROFFS).
9. Append RLEN copies of BIT to **BITS**.
10. Add RLEN to the value LEN. LEN MUST be less than or equal to **NBITS**.
11. If LEN equals **NBITS**, return the completely decoded string **BITS**.
12. Assign BIT the value (1 − BIT).
13. Continue decoding runs from step 5.

7.3 Coded Block Flags Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
FTYPE	Integer	1	No	The frame type.
NSBS	Integer	32	No	The total number of super blocks in a frame.
NBS	Integer	36	No	The total number of blocks in a frame.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
NBITS	Integer	36	No	The length of a bit string to decode.
BITS	Bit string			A decoded set of flags.
SBPCODED	Integer Array	1	No	An **NSBS**-element array of flags indicating whether or not each super block is partially coded.
SBFCODED	Integer Array	1	No	An **NSBS**-element array of flags indicating whether or not each non-partially coded super block is fully coded.
sbi	Integer	32	No	The index of the current super block.
bi	Integer	36	No	The index of the current block in coded order.

This procedure determines which blocks are coded in a given frame. In an intra frame, it marks all blocks coded. In an inter frame, however, any or all of the blocks may remain uncoded. The output is a list of bit flags, one for each block, marking it coded or not coded.

It is important to note that flags are still decoded for any blocks which lie

entirely outside the picture region, even though they are not displayed. Encoders MAY choose to code such blocks. Decoders MUST faithfully reconstruct such blocks, because their contents can be used for predictors in future frames. Flags are *not* decoded for portions of a super block which lie outside the full frame, as there are no blocks in those regions.

The complete procedure is as follows:

1. If **FTYPE** is zero (intra frame):
   (a) For each consecutive value of bi from 0 to (NBS−1), assign **BCODED**[bi] the value one.
2. Otherwise (inter frame):
   (a) Assign NBITS the value **NSBS**.
   (b) Read an NBITS-bit bit string into BITS, using the procedure described in Section 7.2.1. This represents the list of partially coded super blocks.
   (c) For each consecutive value of sbi from 0 to (NSBS − 1), remove the bit at the head of the string BITS and assign it to SBPCODED[sbi].
   (d) Assign NBITS the total number of super blocks such that SBPCODED[sbi] equals zero.
   (e) Read an NBITS-bit bit string into BITS, using the procedure described in Section 7.2.1. This represents the list of fully coded super blocks.
   (f) For each consecutive value of sbi from 0 to (NSBS − 1) such that SBPCODED[sbi] equals zero, remove the bit at the head of the string BITS and assign it to SBFCODED[sbi].
   (g) Assign NBITS the number of blocks contained in super blocks where SBPCODED[sbi] equals one. Note that this might *not* be equal to 16 times the number of partially coded super blocks, since super blocks which overlap the edge of the frame will have fewer than 16 blocks in them.
   (h) Read an NBITS-bit bit string into BITS, using the procedure described in Section 7.2.2.
   (i) For each block in coded order—indexed by bi:
      i. Assign sbi the index of the super block containing block bi.
      ii. If SBPCODED[sbi] is zero, assign **BCODED**[bi] the value SBFCODED[sbi].
      iii. Otherwise, remove the bit at the head of the string BITS and assign it to **BCODED**[bi].

7.4 Macro Block Coding Modes

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
FTYPE	Integer	1	No	The frame type.
NMBS	Integer	32	No	The total number of macro blocks in a frame.
NBS	Integer	36	No	The total number of blocks in a frame.
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
MBMODES	Integer Array	3	No	An **NMBS**-element array of coding modes for each macro block.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
MSCHEME	Integer	3	No	The mode coding scheme.
MALPHABET	Integer array	3	No	The list of modes corresponding to each Huffman code.
mbi	Integer	32	No	The index of the current macro block.
bi	Integer	36	No	The index of the current block in coded order.
mi	Integer	3	No	The index of a Huffman code from Table 7.19, starting from 0.

In an intra frame, every macro block marked as coded in INTRA mode. In an inter frame, however, a macro block can be coded in one of eight coding modes, given in Table 7.18. All of the blocks in all color planes contained in a macro block will be assigned the coding mode of that macro block.

An important thing to note is that a coding mode is only stored in the bitstream for a macro block if it has at least one *luma* block coded. A macro

Index	Coding Mode
0	INTER_NOMV
1	INTRA
2	INTER_MV
3	INTER_MV_LAST
4	INTER_MV_LAST2
5	INTER_GOLDEN_NOMV
6	INTER_GOLDEN_MV
7	INTER_MV_FOUR

Table 7.18: Macro Block Coding Modes

block that contains coded blocks in the chroma planes, but not in the luma plane, MUST be coded in INTER_NOMV mode. Thus, no coding mode needs to be decoded for such a macro block.

Coding modes are encoded using one of eight different schemes. Schemes 0 through 6 use the same simple Huffman code to represent the mode numbers, as given in Table 7.19. The difference in the schemes is the mode number assigned to each code. Scheme 0 uses an assignment specified in the bitstream, while schemes 1–6 use a fixed assignment, also given in Table 7.19. Scheme 7 simply codes each mode directly in the bitstream using three bits.

Scheme	1	2	3	4	5	6	7
Huffman Code	Coding Mode						*mi*
`b0`	3	3	3	3	0	0	0
`b10`	4	4	2	2	3	5	1
`b110`	2	0	4	0	4	3	2
`b1110`	0	2	0	4	2	4	3
`b11110`	1	1	1	1	1	2	4
`b111110`	5	5	5	5	5	1	5
`b1111110`	6	6	6	6	6	6	6
`b1111111`	7	7	7	7	7	7	7

Table 7.19: Macro Block Mode Schemes

1. If **FTYPE** is 0 (intra frame):
   (a) For each consecutive value of *mbi* from 0 to (**NMBS** − 1), inclusive, assign **MBMODES**[*mbi*] the value 1 (INTRA).
2. Otherwise (inter frame):
   (a) Read a 3-bit unsigned integer as MSCHEME.

(b) If MSCHEME is 0:

i. For each consecutive value of MODE from 0 to 7, inclusive:

A. Read a 3-bit unsigned integer as mi.

B. Assign MALPHABET[mi] the value MODE.

(c) Otherwise, if MSCHEME is not 7, assign the entries of MALPHABET the values in the corresponding column of Table 7.19.

(d) For each consecutive macro block in coded order (cf. Section 2.4)—indexed by mbi:

i. If a block bi in the luma plane of macro block mbi exists such that **BCODED**[bi] is 1:

A. If MSCHEME is not 7, read one bit at a time until one of the Huffman codes in Table 7.19 is recognized, and assign **MBMODES**[mbi] the value MALPHABET[mi], where mi is the index of the Huffman code decoded.

B. Otherwise, read a 3-bit unsigned integer as **MBMODES**[mbi].

ii. Otherwise, if no luma-plane blocks in the macro block are coded, assign **MBMODES**[mbi] the value 0 (INTER_NOMV).

7.5 Motion Vectors

In an intra frame, no motion vectors are used, and so motion vector decoding is skipped. In an inter frame, however, many of the inter coding modes require a motion vector in order to specify an offset into the reference frame from which to predict a block. These procedures assigns such a motion vector to every block.

7.5.1 Motion Vector Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
MVMODE	Integer	1	No	The motion vector decoding method.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
MVX	Integer	6	Yes	The X component of the motion vector.
MVY	Integer	6	Yes	The Y component of the motion vector.

Name	Type	Size (bits)	Signed?	Description and restrictions

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
MVSIGN	Integer	1	No	The sign of the motion vector component just decoded.

The individual components of a motion vector can be coded using one of two methods. The first uses a variable length Huffman code, given in Table 7.23. The second encodes the magnitude of the component directly in 5 bits, and the sign in one bit. Note that in this case there are two representations for the value zero. For compatibility with VP3, a sign bit is read even if the magnitude read is zero. One scheme is chosen and used for the entire frame.

Each component can take on integer values from $-31 \ldots 31$, inclusive, at half-pixel resolution, i.e. $-15.5 \ldots 15.5$ pixels in the luma plane. For each subsampled axis in the chroma planes, the corresponding motion vector component is interpreted as being at quarter-pixel resolution, i.e. $-7.75 \ldots 7.75$ pixels. The precise details of how these vectors are used to compute predictors for each block are described in Section 7.9.1.

A single motion vector is decoded is follows:

1. If **MVMODE** is 0:
   (a) Read 1 bit at a time until one of the Huffman codes in Table 7.23 is recognized, and assign the value to MVX.
   (b) Read 1 bit at a time until one of the Huffman codes in Table 7.23 is recognized, and assign the value to MVY.
2. Otherwise:
   (a) Read a 5-bit unsigned integer as **MVX**.
   (b) Read a 1-bit unsigned integer as MVSIGN.
   (c) If MVSIGN is 1, assign **MVX** the value $-$**MVX**.
   (d) Read a 5-bit unsigned integer as **MVY**.
   (e) Read a 1-bit unsigned integer as MVSIGN.
   (f) If MVSIGN is 1, assign **MVY** the value $-$**MVY**.

7.5.2 Macro Block Motion Vector Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
PF	Integer	2	No	The pixel format.
NMBS	Integer	32	No	The total number of macro blocks in a frame.
MBMODES	Integer Array	3	No	An **NMBS**-element array of coding modes for each macro block.
NBS	Integer	36	No	The total number of blocks in a frame.
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
MVECTS	Array of 2D Integer Vectors	6	Yes	An **NBS**-element array of motion vectors for each block.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
LAST1	2D Integer Vector	6	Yes	The last motion vector.
LAST2	2D Integer Vector	6	Yes	The second to last motion vector.
MVX	Integer	6	Yes	The X component of a motion vector.
MVY	Integer	6	Yes	The Y component of a motion vector.
mbi	Integer	32	No	The index of the current macro block.
A	Integer	36	No	The index of the lower-left luma block in the macro block.
B	Integer	36	No	The index of the lower-right luma block in the macro block.
C	Integer	36	No	The index of the upper-left luma block in the macro block.

Name	Type	Size (bits)	Signed?	Description and restrictions
D	Integer	36	No	The index of the upper-right luma block in the macro block.
E	Integer	36	No	The index of a chroma block in the macro block, depending on the pixel format.
F	Integer	36	No	The index of a chroma block in the macro block, depending on the pixel format.
G	Integer	36	No	The index of a chroma block in the macro block, depending on the pixel format.
H	Integer	36	No	The index of a chroma block in the macro block, depending on the pixel format.
I	Integer	36	No	The index of a chroma block in the macro block, depending on the pixel format.
J	Integer	36	No	The index of a chroma block in the macro block, depending on the pixel format.
K	Integer	36	No	The index of a chroma block in the macro block, depending on the pixel format.
L	Integer	36	No	The index of a chroma block in the macro block, depending on the pixel format.

Motion vectors are stored for each macro block. In every mode except for INTER_MV_FOUR, every block in all the color planes are assigned the same motion vector. In INTER_MV_FOUR mode, all four blocks in the luma plane are assigned their own motion vector, and motion vectors for blocks in the chroma planes are computed from these, using averaging appropriate to the pixel format.

For INTER_MV and INTER_GOLDEN_MV modes, a single motion vector is decoded and applied to each block. For INTER_MV_FOUR macro blocks, a motion vector is decoded for each coded luma block. Uncoded luma blocks receive the default $(0, 0)$ vector for the purposes of computing the chroma motion vectors.

None of the remaining macro block coding modes require decoding motion vectors from the stream. INTRA mode does not use a motion-compensated predictor, and so requires no motion vector, and INTER_NOMV and INTER_GOLDEN_NOMV modes use the default vector $(0, 0)$ for each block. This also includes all macro

blocks with no coded luma blocks, as they are coded in INTER_NOMV mode by definition.

The modes INTER_MV_LAST and INTER_MV_LAST2 use the motion vector from the last macro block (in coded order) and the second to last macro block, respectively, that contained a motion vector pointing to the previous frame. Thus no explicit motion vector needs to be decoded for these modes. Macro blocks coded in INTRA mode or one of the GOLDEN modes are not considered in this process. If an insufficient number of macro blocks have been coded in one of the INTER modes, then the $(0, 0)$ vector is used instead. For macro blocks coded in INTER_MV_FOUR mode, the vector from the upper-right luma block is used, even if the upper-right block is not coded.

The motion vectors are decoded from the stream as follows:

1. Assign LAST1 and LAST2 both the value $(0, 0)$.
2. Read a 1-bit unsigned integer as MVMODE. Note that this value is read even if no macro blocks require a motion vector to be decoded.
3. For each consecutive value of mbi from 0 to (**NMBS** $- 1$):
   (a) If **MBMODES**[mbi] is 7 (INTER_MV_FOUR):
      i. Let A, B, C, and D be the indices in coded order bi of the luma blocks in macro block mbi, arranged into raster order. Thus, A is the index in coded order of the block in the lower left, B the lower right, C the upper left, and D the upper right.
      ii. If **BCODED**[A] is non-zero:
         A. Decode a single motion vector into MVX and MVY using the procedure described in Section 7.5.1.
         B. Assign **MVECTS**[A] the value (MVX, MVY).
      iii. Otherwise, assign **MVECTS**[A] the value $(0, 0)$.
      iv. If **BCODED**[B] is non-zero:
         A. Decode a single motion vector into MVX and MVY using the procedure described in Section 7.5.1.
         B. Assign **MVECTS**[B] the value (MVX, MVY).
      v. Otherwise assign **MVECTS**[B] the value $(0, 0)$.
      vi. If **BCODED**[C] is non-zero:
         A. Decode a single motion vector into MVX and MVY using the procedure described in Section 7.5.1.
         B. Assign **MVECTS**[C] the value (MVX, MVY).
      vii. Otherwise assign **MVECTS**[C] the value $(0, 0)$.
      viii. If **BCODED**[D] is non-zero:
         A. Decode a single motion vector into MVX and MVY using the procedure described in Section 7.5.1.
         B. Assign **MVECTS**[D] the value (MVX, MVY).

ix. Otherwise, assign **MVECTS**[D] the value $(0, 0)$.

x. If **PF** is 0 (4:2:0):

A. Let E and F be the index in coded order of the one block in the macro block from the C_b and C_r planes, respectively.

B. Assign **MVECTS**[E] and **MVECTS**[F] the value

$$(\text{round}\left(\frac{\begin{array}{c}\mathbf{MVECTS}[\text{A}]_x + \mathbf{MVECTS}[\text{B}]_x + \\ \mathbf{MVECTS}[\text{C}]_x + \mathbf{MVECTS}[\text{D}]_x\end{array}}{4}\right), \text{round}\left(\frac{\begin{array}{c}\mathbf{MVECTS}[\text{A}]_y + \mathbf{MVECTS}[\text{B}]_y + \\ \mathbf{MVECTS}[\text{C}]_y + \mathbf{MVECTS}[\text{D}]_y\end{array}}{4}\right))$$

xi. If **PF** is 2 (4:2:2):

A. Let E and F be the indices in coded order of the bottom and top blocks in the macro block from the C_b plane, respectively, and G and H be the indices in coded order of the bottom and top blocks in the C_r plane, respectively.

B. Assign **MVECTS**[E] and **MVECTS**[G] the value

$$(\text{round}\left(\frac{\mathbf{MVECTS}[\text{A}]_x + \mathbf{MVECTS}[\text{B}]_x}{2}\right), \text{round}\left(\frac{\mathbf{MVECTS}[\text{A}]_y + \mathbf{MVECTS}[\text{B}]_y}{2}\right))$$

C. Assign **MVECTS**[F] and **MVECTS**[H] the value

$$(\text{round}\left(\frac{\mathbf{MVECTS}[\text{C}]_x + \mathbf{MVECTS}[\text{D}]_x}{2}\right), \text{round}\left(\frac{\mathbf{MVECTS}[\text{C}]_y + \mathbf{MVECTS}[\text{D}]_y}{2}\right))$$

xii. If **PF** is 3 (4:4:4):

A. Let E, F, G, and H be the indices bi in coded order of the C_b plane blocks in macro block mbi, arranged into raster order, and I, J, K, and L be the indices bi in coded order of the C_r plane blocks in macro block mbi, arranged into raster order.

B. Assign **MVECTS**[E] and **MVECTS**[I] the value **MVECTS**[A].

C. Assign **MVECTS**[F] and **MVECTS**[J] the value **MVECTS**[B].

D. Assign **MVECTS**[G] and **MVECTS**[K] the value **MVECTS**[C].

E. Assign **MVECTS**[H] and **MVECTS**[L] the value **MVECTS**[D].

xiii. Assign LAST2 the value LAST1.

xiv. Assign LAST1 the value (MVX, MVY). This is the value of the motion vector decoded from the last coded luma block in raster order. There must always be at least one, since macro blocks with no coded luma blocks must use mode 0: INTER_NOMV.

(b) Otherwise, if **MBMODES**[*mbi*] is 6 (INTER_GOLDEN_MV), decode a single motion vector into MVX and MVY using the procedure described in Section 7.5.1.

(c) Otherwise, if **MBMODES**[*mbi*] is 4 (INTER_MV_LAST2):

i. Assign (MVX, MVY) the value LAST2.

ii. Assign LAST2 the value LAST1.

iii. Assign LAST1 the value (MVX, MVY).

(d) Otherwise, if **MBMODES**[*mbi*] is 3 (INTER_MV_LAST), assign (MVX, MVY) the value LAST1.

(e) Otherwise, if **MBMODES**[*mbi*] is 2 (INTER_MV):

i. Decode a single motion vector into MVX and MVY using the procedure described in Section 7.5.1.

ii. Assign LAST2 the value LAST1.

iii. Assign LAST1 the value (MVX, MVY).

(f) Otherwise (**MBMODES**[*mbi*] is 5: INTER_GOLDEN_NOMV, 1: INTRA, or 0: INTER_NOMV), assign MVX and MVY the value zero.

(g) If **MBMODES**[*mbi*] is not 7 (not INTER_MV_FOUR), then for each coded block *bi* in macro block *mbi*:

i. Assign **MVECTS**[*bi*] the value (MVX, MVY).

VP3 Compatibility Unless all four luma blocks in the macro block are coded, the VP3 encoder does not select mode INTER_MV_FOUR. Theora removes this restriction by treating the motion vector for an uncoded luma block as the default $(0, 0)$ vector. This is consistent with the premise that the block has not changed since the previous frame and that chroma information can be largely ignored when estimating motion.

No modification is required for INTER_MV_FOUR macro blocks in VP3 streams to be decoded correctly by a Theora decoder. However, regardless of how many of the luma blocks are actually coded, the VP3 decoder always reads four motion vectors from the stream for INTER_MV_FOUR mode. The motion vectors read are used to calculate the motion vectors for the chroma blocks, but are otherwise ignored. Thus, care should be taken when creating Theora streams meant to be backwards compatible with VP3 to only use INTER_MV_FOUR mode when all four luma blocks are coded.

7.6 Block-Level *qi* Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
NBS	Integer	36	No	The total number of blocks in a frame.
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.
NQIS	Integer	2	No	The number of *qi* values.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
QIIS	Integer Array	2	No	An **NBS**-element array of *qii* values for each block.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
NBITS	Integer	36	No	The length of a bit string to decode.
BITS	Bit string			A decoded set of flags.
bi	Integer	36	No	The index of the current block in coded order.
qii	Integer	2	No	The index of *qi* value in the list of *qi* values defined for this frame.

This procedure selects the *qi* value to be used for dequantizing the AC coefficients of each block. DC coefficients all use the same *qi* value, so as to avoid interference with the DC prediction mechanism, which occurs in the quantized domain.

The value is actually represented by an index *qii* into the list of *qi* values defined for the frame. The decoder makes multiple passes through the list of coded blocks, one for each *qi* value except the last one. In each pass, an RLE-coded bitmask is decoded to divide the blocks into two groups: those that use the current *qi* value in the list, and those that use a value from later in the list.

Each subsequent pass is restricted to the blocks in the second group.

1. For each value of bi from 0 to (**NBS** -1), assign **QIIS**$[bi]$ the value zero.
2. For each consecutive value of qii from 0 to (**NQIS** -2):
   (a) Assign NBITS be the number of blocks bi such that **BCODED**$[bi]$ is non-zero and **QIIS**$[bi]$ equals qii.
   (b) Read an NBITS-bit bit string into BITS, using the procedure described in Section 7.2.1. This represents the list of blocks that use qi value qii or higher.
   (c) For each consecutive value of bi from 0 to (**NBS** -1) such that **BCODED**$[bi]$ is non-zero and **QIIS**$[bi]$ equals qii:
      i. Remove the bit at the head of the string BITS and add its value to **QIIS**$[bi]$.

VP3 Compatibility For VP3 compatible streams, only one qi value can be specified in the frame header, so the main loop of the above procedure, which would iterate from 0 to -1, is never executed. Thus, no bits are read, and each block uses the one qi value defined for the frame.

Huffman Code	Value	Huffman Code	Value
b000	0		
b001	1	b010	−1
b0110	2	b0111	−2
b1000	3	b1001	−3
b101000	4	b101001	−4
b101010	5	b101011	−5
b101100	6	b101101	−6
b101110	7	b101111	−7
b1100000	8	b1100001	−8
b1100010	9	b1100011	−9
b1100100	10	b1100101	−10
b1100110	11	b1100111	−11
b1101000	12	b1101001	−12
b1101010	13	b1101011	−13
b1101100	14	b1101101	−14
b1101110	15	b1101111	−15
b11100000	16	b11100001	−16
b11100010	17	b11100011	−17
b11100100	18	b11100101	−18
b11100110	19	b11100111	−19
b11101000	20	b11101001	−20
b11101010	21	b11101011	−21
b11101100	22	b11101101	−22
b11101110	23	b11101111	−23
b11110000	24	b11110001	−24
b11110010	25	b11110011	−25
b11110100	26	b11110101	−26
b11110110	27	b11110111	−27
b11111000	28	b11111001	−28
b11111010	29	b11111011	−29
b11111100	30	b11111101	−30
b11111110	31	b11111111	−31

Table 7.23: Huffman Codes for Motion Vector Components

7.7 DCT Coefficients

The quantized DCT coefficients are decoded by making 64 passes through the list of coded blocks, one for each token index in zig-zag order. For the DC tokens, two Huffman tables are chosen from among the first 16, one for the luma plane and one for the chroma planes. The AC tokens, however, are divided into four different groups. Again, two 4-bit indices are decoded, one for the luma plane, and one for the chroma planes, but these select the codebooks for *all four* groups. AC coefficients in group one use codebooks $16 \ldots 31$, while group two uses $32 \ldots 47$, etc. Note that this second set of indices is decoded even if there are no non-zero AC coefficients in the frame.

Tokens are divided into two major types: EOB tokens, which fill the remainder of one or more blocks with zeros, and coefficient tokens, which fill in one or more coefficients within a single block. A decoding procedure for the first is given in Section 7.7.1, and for the second in Section 7.7.2. The decoding procedure for the complete set of quantized coefficients is given in Section 7.7.3.

7.7.1 EOB Token Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
TOKEN	Integer	5	No	The token being decoded. This must be in the range $0 \ldots 6$.
NBS	Integer	36	No	The total number of blocks in a frame.
TIS	Integer Array	7	No	An **NBS**-element array of the current token index for each block.
NCOEFFS	Integer Array	7	No	An **NBS**-element array of the coefficient count for each block.
COEFFS	2D Integer Array	16	Yes	An **NBS** $\times$ 64 array of quantized DCT coefficient values for each block in zig-zag order.
bi	Integer	36	No	The index of the current block in coded order.
ti	Integer	6	No	The current token index.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
TIS	Integer Array	7	No	An **NBS**-element array of the current token index for each block.
COEFFS	2D Integer Array	16	Yes	An **NBS**×64 array of quantized DCT coefficient values for each block in zig-zag order.
EOBS	Integer	36	No	The remaining length of the current EOB run.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
bj	Integer	36	No	Another index of a block in coded order.
tj	Integer	6	No	Another token index.

A summary of the EOB tokens is given in Table 7.33. An important thing to note is that token 6 does not add an offset to the decoded run value, even though in general it should only be used for runs of size 32 or longer. If a value of zero is decoded for this run, it is treated as an EOB run the size of the remaining coded blocks.

Token Value	Extra Bits	EOB Run Lengths
0	0	1
1	0	2
2	0	3
3	2	4 . . . 7
4	3	8 . . . 15
5	4	16 . . . 31
6	12	1 . . . 4095, or all remaining blocks

Table 7.33: EOB Token Summary

There is no restriction that one EOB token cannot be immediately followed by another, so no special cases are necessary to extend the range of the maximum run length as were required in Section 7.2.1. Indeed, depending on the lengths of the Huffman codes, it may even cheaper to encode, by way of example, an EOB run of length 31 followed by an EOB run of length 1 than to encode an EOB run of length 32 directly. There is also no restriction that an EOB run

stop at the end of a color plane or a token index. The run MUST, however, end at or before the end of the frame.

1. If **TOKEN** is 0, assign **EOBS** the value 1.
2. Otherwise, if **TOKEN** is 1, assign **EOBS** the value 2.
3. Otherwise, if **TOKEN** is 2, assign **EOBS** the value 3.
4. Otherwise, if **TOKEN** is 3:
   (a) Read a 2-bit unsigned integer as **EOBS**.
   (b) Assign **EOBS** the value (**EOBS** + 4).
5. Otherwise, if **TOKEN** is 4:
   (a) Read a 3-bit unsigned integer as **EOBS**.
   (b) Assign **EOBS** the value (**EOBS** + 8).
6. Otherwise, if **TOKEN** is 5:
   (a) Read a 4-bit unsigned integer as **EOBS**.
   (b) Assign **EOBS** the value (**EOBS** + 16).
7. Otherwise, **TOKEN** is 6:
   (a) Read a 12-bit unsigned integer as **EOBS**.
   (b) If **EOBS** is zero, assign **EOBS** to be the number of coded blocks *bj* such that **TIS**[*bj*] is less than 64.
8. For each value of *tj* from ***ti*** to 63, assign **COEFFS**[***bi***][*tj*] the value zero.
9. Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].
10. Assign **TIS**[***bi***] the value 64.
11. Assign **EOBS** the value (**EOBS** − 1).

VP3 Compatibility The VP3 encoder does not use the special interpretation of a zero-length EOB run, though its decoder *does* support it. That may be due more to a happy accident in the way the decoder was written than intentional design, however, and other VP3 implementations might not reproduce it faithfully. For backwards compatibility, it may be wise to avoid it, especially as for most frame sizes there are fewer than 4095 blocks, making it unnecessary.

7.7.2 Coefficient Token Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
TOKEN	Integer	5	No	The token being decoded. This must be in the range $7 \ldots 31$.
NBS	Integer	36	No	The total number of blocks in a frame.
TIS	Integer Array	7	No	An **NBS**-element array of the current token index for each block.
COEFFS	2D Integer Array	16	Yes	An **NBS**$\times 64$ array of quantized DCT coefficient values for each block in zig-zag order.
bi	Integer	36	No	The index of the current block in coded order.
ti	Integer	6	No	The current token index.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
TIS	Integer Array	7	No	An **NBS**-element array of the current token index for each block.
NCOEFFS	Integer Array	7	No	An **NBS**-element array of the coefficient count for each block.
COEFFS	2D Integer Array	16	Yes	An **NBS** $\times$ 64 array of quantized DCT coefficient values for each block in zig-zag order.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
SIGN	Integer	1	No	A flag indicating the sign of the current coefficient.
MAG	Integer	10	No	The magnitude of the current coefficient.
RLEN	Integer	6	No	The length of the current zero run.
tj	Integer	6	No	Another token index.

Each of these tokens decodes one or more coefficients in the current block. A summary of the meanings of the token values is presented in Table 7.38. There are often several different ways to tokenize a given coefficient list. Which one is optimal depends on the exact lengths of the Huffman codes used to represent each token. Note that we do not update the coefficient count for the block if we decode a pure zero run.

Token Value	Extra Bits	Number of Coefficients	Description
7	3	1...8	Short zero run.
8	6	1...64	Zero run.
9	0	1	1.
10	0	1	−1.
11	0	1	2.
12	0	1	−2.
13	1	1	±3.
14	1	1	±4.
15	1	1	±5.
16	1	1	±6.
17	2	1	±7...8.
18	3	1	±9...12.
19	4	1	±13...20.
20	5	1	±21...36.
21	6	1	±37...68.
22	10	1	±69...580.
23	1	2	One zero followed by ±1.
24	1	3	Two zeros followed by ±1.
25	1	4	Three zeros followed by ±1.
26	1	5	Four zeros followed by ±1.
27	1	6	Five zeros followed by ±1.
28	3	7...10	6...9 zeros followed by ±1.
29	4	11...18	10...17 zeros followed by ±1.
30	2	2	One zero followed by ±2...3.
31	3	3...4	2...3 zeros followed by ±2...3.

Table 7.38: Coefficient Token Summary

For tokens which represent more than one coefficient, they MUST NOT bring the total number of coefficients in the block to more than 64. Care should be taken in a decoder to check for this, as otherwise it may permit buffer overflows from invalidly formed packets.

Note: One way to achieve this efficiently is to combine the inverse zig-zag mapping (described later in Section 7.9.2) with coefficient

decode, and use a table look-up to map zig-zag indices greater than 63 to a safe location.

1. If **TOKEN** is 7:
   (a) Read in a 3-bit unsigned integer as RLEN.
   (b) Assign RLEN the value (RLEN + 1).
   (c) For each value of *tj* from ***ti*** to (***ti***+RLEN−1), assign **COEFFS**[***bi***][*tj*] the value zero.
   (d) Assign **TIS**[***bi***] the value **TIS**[***bi***] + RLEN.
2. Otherwise, if **TOKEN** is 8:
   (a) Read in a 6-bit unsigned integer as RLEN.
   (b) Assign RLEN the value (RLEN + 1).
   (c) For each value of *tj* from ***ti*** to (***ti***+RLEN−1), assign **COEFFS**[***bi***][*tj*] the value zero.
   (d) Assign **TIS**[***bi***] the value **TIS**[***bi***] + RLEN.
3. Otherwise, if **TOKEN** is 9:
   (a) Assign **COEFFS**[***bi***][***ti***] the value 1.
   (b) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.
   (c) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].
4. Otherwise, if **TOKEN** is 10:
   (a) Assign **COEFFS**[***bi***][***ti***] the value −1.
   (b) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.
   (c) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].
5. Otherwise, if **TOKEN** is 11:
   (a) Assign **COEFFS**[***bi***][***ti***] the value 2.
   (b) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.
   (c) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].
6. Otherwise, if **TOKEN** is 12:
   (a) Assign **COEFFS**[***bi***][***ti***] the value −2.
   (b) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.
   (c) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].
7. Otherwise, if **TOKEN** is 13:
   (a) Read a 1-bit unsigned integer as SIGN.

(b) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value 3.

(c) Otherwise, assign **COEFFS**[***bi***][***ti***] the value -3.

(d) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(e) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

8. Otherwise, if **TOKEN** is 14:

(a) Read a 1-bit unsigned integer as SIGN.

(b) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value 4.

(c) Otherwise, assign **COEFFS**[***bi***][***ti***] the value -4.

(d) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(e) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

9. Otherwise, if **TOKEN** is 15:

(a) Read a 1-bit unsigned integer as SIGN.

(b) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value 5.

(c) Otherwise, assign **COEFFS**[***bi***][***ti***] the value -5.

(d) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(e) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

10. Otherwise, if **TOKEN** is 16:

(a) Read a 1-bit unsigned integer as SIGN.

(b) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value 6.

(c) Otherwise, assign **COEFFS**[***bi***][***ti***] the value -6.

(d) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(e) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

11. Otherwise, if **TOKEN** is 17:

(a) Read a 1-bit unsigned integer as SIGN.

(b) Read a 1-bit unsigned integer as MAG.

(c) Assign MAG the value (MAG + 7).

(d) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value MAG.

(e) Otherwise, assign **COEFFS**[***bi***][***ti***] the value $-$MAG.

(f) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(g) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

12. Otherwise, if **TOKEN** is 18:

(a) Read a 1-bit unsigned integer as SIGN.

(b) Read a 2-bit unsigned integer as MAG.

(c) Assign MAG the value (MAG + 9).

(d) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value MAG.

(e) Otherwise, assign **COEFFS**[***bi***][***ti***] the value −MAG.

(f) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(g) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

13. Otherwise, if **TOKEN** is 19:

(a) Read a 1-bit unsigned integer as SIGN.

(b) Read a 3-bit unsigned integer as MAG.

(c) Assign MAG the value (MAG + 13).

(d) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value MAG.

(e) Otherwise, assign **COEFFS**[***bi***][***ti***] the value −MAG.

(f) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(g) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

14. Otherwise, if **TOKEN** is 20:

(a) Read a 1-bit unsigned integer as SIGN.

(b) Read a 4-bit unsigned integer as MAG.

(c) Assign MAG the value (MAG + 21).

(d) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value MAG.

(e) Otherwise, assign **COEFFS**[***bi***][***ti***] the value −MAG.

(f) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(g) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

15. Otherwise, if **TOKEN** is 21:

(a) Read a 1-bit unsigned integer as SIGN.

(b) Read a 5-bit unsigned integer as MAG.

(c) Assign MAG the value (MAG + 37).

(d) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value MAG.

(e) Otherwise, assign **COEFFS**[***bi***][***ti***] the value −MAG.

(f) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(g) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

16. Otherwise, if **TOKEN** is 22:

(a) Read a 1-bit unsigned integer as SIGN.

(b) Read a 9-bit unsigned integer as MAG.

(c) Assign MAG the value (MAG + 69).

(d) If SIGN is zero, assign **COEFFS**[***bi***][***ti***] the value MAG.

(e) Otherwise, assign **COEFFS**[***bi***][***ti***] the value −MAG.

(f) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 1.

(g) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

17. Otherwise, if **TOKEN** is 23:

(a) Assign **COEFFS**[***bi***][***ti***] the value zero.

(b) Read a 1-bit unsigned integer as SIGN.

(c) If SIGN is zero, assign **COEFFS**[***bi***][***ti*** + 1] the value 1.

(d) Otherwise, assign **COEFFS**[***bi***][***ti*** + 1] the value −1.

(e) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 2.

(f) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

18. Otherwise, if **TOKEN** is 24:

(a) For each value of *tj* from ***ti*** to (***ti*** + 1), assign **COEFFS**[***bi***][*tj*] the value zero.

(b) Read a 1-bit unsigned integer as SIGN.

(c) If SIGN is zero, assign **COEFFS**[***bi***][***ti*** + 2] the value 1.

(d) Otherwise, assign **COEFFS**[***bi***][***ti*** + 2] the value −1.

(e) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 3.

(f) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

19. Otherwise, if **TOKEN** is 25:

(a) For each value of *tj* from ***ti*** to (***ti*** + 2), assign **COEFFS**[***bi***][*tj*] the value zero.

(b) Read a 1-bit unsigned integer as SIGN.

(c) If SIGN is zero, assign **COEFFS**[***bi***][***ti*** + 3] the value 1.

(d) Otherwise, assign **COEFFS**[***bi***][***ti*** + 3] the value −1.

(e) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 4.

(f) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

20. Otherwise, if **TOKEN** is 26:

(a) For each value of *tj* from ***ti*** to (***ti*** + 3), assign **COEFFS**[***bi***][*tj*] the value zero.

(b) Read a 1-bit unsigned integer as SIGN.

(c) If SIGN is zero, assign **COEFFS**[***bi***][***ti*** + 4] the value 1.

(d) Otherwise, assign **COEFFS**[***bi***][***ti*** + 4] the value −1.

(e) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 5.

(f) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

21. Otherwise, if **TOKEN** is 27:

(a) For each value of *tj* from ***ti*** to (***ti*** + 4), assign **COEFFS**[***bi***][*tj*] the value zero.

(b) Read a 1-bit unsigned integer as SIGN.

(c) If SIGN is zero, assign **COEFFS**[***bi***][***ti*** + 5] the value 1.

(d) Otherwise, assign **COEFFS**[***bi***][***ti*** + 5] the value −1.

(e) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 6.

(f) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

22. Otherwise, if **TOKEN** is 28:

(a) Read a 1-bit unsigned integer as SIGN.

(b) Read a 2-bit unsigned integer as RLEN.

(c) Assign RLEN the value (RLEN + 6).

(d) For each value of *tj* from ***ti*** to (***ti***+RLEN−1), assign **COEFFS**[***bi***][*tj*] the value zero.

(e) If SIGN is zero, assign **COEFFS**[***bi***][***ti*** + RLEN] the value 1.

(f) Otherwise, assign **COEFFS**[***bi***][***ti*** + RLEN] the value −1.

(g) Assign **TIS**[***bi***] the value **TIS**[***bi***] + RLEN + 1.

(h) Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

23. Otherwise, if **TOKEN** is 29:

(a) Read a 1-bit unsigned integer as SIGN.

(b) Read a 3-bit unsigned integer as RLEN.

(c) Assign RLEN the value (RLEN + 10).

(d) For each value of *tj* from ***ti*** to (***ti***+RLEN−1), assign **COEFFS**[***bi***][*tj*] the value zero.

(e) If SIGN is zero, assign **COEFFS**[***bi***][***ti*** + RLEN] the value 1.

(f) Otherwise, assign **COEFFS**[***bi***][***ti*** + RLEN] the value −1.

(g) Assign **TIS**[***bi***] the value **TIS**[***bi***]+RLEN+1. Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

24. Otherwise, if **TOKEN** is 30:

(a) Assign **COEFFS**[***bi***][*ti*] the value zero.

(b) Read a 1-bit unsigned integer as SIGN.

(c) Read a 1-bit unsigned integer as MAG.

(d) Assign MAG the value (MAG + 2).

(e) If SIGN is zero, assign **COEFFS**[***bi***][***ti*** + 1] the value MAG.

(f) Otherwise, assign **COEFFS**[***bi***][***ti*** + 1] the value −MAG.

(g) Assign **TIS**[***bi***] the value **TIS**[***bi***] + 2. Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

25. Otherwise, if **TOKEN** is 31:

(a) Read a 1-bit unsigned integer as SIGN.

(b) Read a 1-bit unsigned integer as MAG.

(c) Assign MAG the value (MAG + 2).

(d) Read a 1-bit unsigned integer as RLEN.

(e) Assign RLEN the value (RLEN + 2).

(f) For each value of *tj* from ***ti*** to (***ti***+RLEN−1), assign **COEFFS**[***bi***][*tj*] the value zero.

(g) If SIGN is zero, assign **COEFFS**[***bi***][***ti*** + RLEN] the value MAG.

(h) Otherwise, assign **COEFFS**[***bi***][***ti*** + RLEN] the value −MAG.

(i) Assign **TIS**[***bi***] the value **TIS**[***bi***]+RLEN+1. Assign **NCOEFFS**[***bi***] the value **TIS**[***bi***].

7.7.3 DCT Coefficient Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
NBS	Integer	36	No	The total number of blocks in a frame.
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.
NMBS	Integer	32	No	The total number of macro blocks in a frame.
HTS	Huffman table array			An 80-element array of Huffman tables with up to 32 entries each.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
COEFFS	2D Integer Array	16	Yes	An **NBS** $\times$ 64 array of quantized DCT coefficient values for each block in zig-zag order.
NCOEFFS	Integer Array	7	No	An **NBS**-element array of the coefficient count for each block.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
NLBS	Integer	34	No	The number of blocks in the luma plane.
TIS	Integer Array	7	No	An **NBS**-element array of the current token index for each block.
EOBS	Integer	36	No	The remaining length of the current EOB run.
TOKEN	Integer	5	No	The current token being decoded.
HG	Integer	3	No	The current Huffman table group.
cbi	Integer	36	No	The index of the current block in the coded block list.
bi	Integer	36	No	The index of the current block in coded order.
bj	Integer	36	No	Another index of a block in coded order.
ti	Integer	6	No	The current token index.
tj	Integer	6	No	Another token index.
hti_{L}	Integer	4	No	The index of the current Huffman table to use for the luma plane within a group.
hti_{C}	Integer	4	No	The index of the current Huffman table to use for the chroma planes within a group.
hti	Integer	7	No	The index of the current Huffman table to use.

This procedure puts the above two procedures to work to decode the entire set of DCT coefficients for the frame. At the end of this procedure, EOBS MUST be zero, and TIS[*bi*] MUST be 64 for every coded *bi*.

Note that we update the coefficient count of every block before continuing an EOB run or decoding a token, despite the fact that it is already up to date unless the previous token was a pure zero run. This is done intentionally to mimic the VP3 accounting rules. Thus the only time the coefficient count does not include the coefficients in a pure zero run is when when that run reaches all the way to coefficient 63. Note, however, that regardless of the coefficient count, any additional coefficients are still set to zero. The only use of the count is in determining if a special case of the inverse DCT can be used in Section 7.9.3.

1. Assign NLBS the value (**NMBS** $*$ 4).
2. For each consecutive value of bi from 0 to (**NBS** $-$ 1), assign TIS[bi] the value zero.
3. Assign EOBS the value 0.
4. For each consecutive value of ti from 0 to 63:
   (a) If ti is 0 or 1:
      i. Read a 4-bit unsigned integer as hti_{L}.
      ii. Read a 4-bit unsigned integer as hti_{C}.
   (b) For each consecutive value of bi from 0 to (**NBS** $-$ 1) for which **BCODED**[bi] is non-zero and TIS[bi] equals ti:
      i. Assign **NCOEFFS**[bi] the value ti.
      ii. If EOBS is greater than zero:
         A. For each value of tj from ti to 63, assign **COEFFS**[bi][tj] the value zero.
         B. Assign TIS[bi] the value 64.
         C. Assign EOBS the value (EOBS $-$ 1).
      iii. Otherwise:
         A. Assign HG a value based on ti from Table 7.42.

ti	HG
0	0
1 . . . 5	1
6 . . . 14	2
15 . . . 27	3
28 . . . 63	4

Table 7.42: Huffman Table Groups

         B. If bi is less than NLBS, assign hti the value $(16 * \mathrm{HG} + hti_{\mathrm{L}})$.
         C. Otherwise, assign hti the value $(16 * \mathrm{HG} + hti_{\mathrm{C}})$.
         D. Read one bit at a time until one of the codes in **HTS**[hti] is recognized, and assign the value to TOKEN.

E. If TOKEN is less than 7, expand an EOB token using the procedure given in Section 7.7.1 to update TIS[bi], **COEFFS**[bi], and EOBS.

F. Otherwise, expand a coefficient token using the procedure given in Section 7.7.2 to update TIS[bi], **COEFFS**[bi], and **NCOEFFS**[bi].

7.8 Undoing DC Prediction

The actual value of a DC coefficient decoded by Section 7.7 is the residual from a predicted value computed by the encoder. This prediction is only applied to DC coefficients. Quantized AC coefficients are encoded directly.

This section describes how to undo this prediction to recover the original DC coefficients. The predicted DC value for a block is computed from the DC values of its immediate neighbors which precede the block in raster order. Thus, reversing this prediction must procede in raster order, instead of coded order.

Note that this step comes before dequantizing the coefficients. For this reason, DC coefficients are all quantized with the same qi value, regardless of the block-level qi values decoded in Section 7.6. Those qi values are applied only to the AC coefficients.

7.8.1 Computing the DC Predictor

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.
MBMODES	Integer Array	3	No	An **NMBS**-element array of coding modes for each macro block.
LASTDC	Integer Array	16	Yes	A 3-element array containing the most recently decoded DC value, one for inter mode and for each reference frame.
COEFFS	2D Integer Array	16	Yes	An **NBS** $\times$ 64 array of quantized DCT coefficient values for each block in zig-zag order.
bi	Integer	36	No	The index of the current block in coded order.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
DCPRED	Integer	16	Yes	The predicted DC value for the current block.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
P	Integer Array	1	No	A 4-element array indicating which neighbors can be used for DC prediction.
PBI	Integer Array	36	No	A 4-element array containing the coded-order block index of the current block's neighbors.
W	Integer Array	7	Yes	A 4-element array of the weights to apply to each neighboring DC value.
PDIV	Integer	8	No	The valud to divide the weighted sum by.
bj	Integer	36	No	The index of a neighboring block in coded order.
mbi	Integer	32	No	The index of the macro block containing block *bi*.
mbi	Integer	32	No	The index of the macro block containing block *bj*.
rfi	Integer	2	No	The index of the reference frame indicated by the coding mode for macro block *mbi*.

This procedure outlines how a predictor is formed for a single block.

The predictor is computed as a weighted sum of the neighboring DC values from coded blocks which use the same reference frame. This latter condition is determined only by checking the coding mode for the block. Even if the golden frame and the previous frame are in fact the same, e.g. for the first inter frame after an intra frame, they are still treated as being different for the purposes of DC prediction. The weighted sum is divided by a power of two, with truncation towards zero, and the result is checked for outranging if necessary.

If there are no neighboring coded blocks which use the same reference frame as the current block, then the most recent DC value of any block that used that

reference frame is used instead. If no such block exists, then the predictor is set to zero.

1. Assign *mbi* the index of the macro block containing block ***bi***.
2. Assign *rfi* the value of the Reference Frame Index column of Table 7.46 corresponding to **MBMODES**[*mbi*].

Coding Mode	Reference Frame Index
0 (INTER_NOMV)	1 (Previous)
1 (INTRA)	0 (None)
2 (INTER_MV)	1 (Previous)
3 (INTER_MV_LAST)	1 (Previous)
4 (INTER_MV_LAST2)	1 (Previous)
5 (INTER_GOLDEN_NOMV)	2 (Golden)
6 (INTER_GOLDEN_MV)	2 (Golden)
7 (INTER_MV_FOUR)	1 (Previous)

Table 7.46: Reference Frames for Each Coding Mode

3. If block *bi* is not along the left edge of the coded frame:
   (a) Assign *bj* the coded-order index of block *bi*'s left neighbor, i.e., in the same row but one column to the left.
   (b) If **BCODED**[*bj*] is not zero:
      i. Assign *mbj* the index of the macro block containing block *bj*.
      ii. If the value of the Reference Frame Index column of Table 7.46 corresonding to **MBMODES**[*mbj*] equals *rfi*:
         A. Assign P[0] the value 1.
         B. Assign PBI[0] the value *bj*.
      iii. Otherwise, assign P[0] the value zero.
   (c) Otherwise, assign P[0] the value zero.
4. Otherwise, assign P[0] the value zero.
5. If block *bi* is not along the left edge nor the bottom edge of the coded frame:
   (a) Assign *bj* the coded-order index of block *bi*'s lower-left neighbor, i.e., one row down and one column to the left.
   (b) If **BCODED**[*bj*] is not zero:
      i. Assign *mbj* the index of the macro block containing block *bj*.
      ii. If the value of the Reference Frame Index column of Table 7.46 corresonding to **MBMODES**[*mbj*] equals *rfi*:

A. Assign P[1] the value 1.
B. Assign PBI[1] the value bj.

iii. Otherwise, assign P[1] the value zero.

(c) Otherwise, assign P[1] the value zero.

6. Otherwise, assign P[1] the value zero.

7. If block bi is not along the the bottom edge of the coded frame:

(a) Assign bj the coded-order index of block bi's lower neighbor, i.e., in the same column but one row down.

(b) If **BCODED**[bj] is not zero:

i. Assign mbj the index of the macro block containing block bj.

ii. If the value of the Reference Frame Index column of Table 7.46 corresonding to **MBMODES**[mbj] equals rfi:

A. Assign P[2] the value 1.
B. Assign PBI[2] the value bj.

iii. Otherwise, assign P[2] the value zero.

(c) Otherwise, assign P[2] the value zero.

8. Otherwise, assign P[2] the value zero.

9. If block bi is not along the right edge nor the bottom edge of the coded frame:

(a) Assign bj the coded-order index of block bi's lower-right neighbor, i.e., one row down and one column to the right.

(b) If **BCODED**[bj] is not zero:

i. Assign mbj the index of the macro block containing block bj.

ii. If the value of the Reference Frame Index column of Table 7.46 corresonding to **MBMODES**[mbj] equals rfi:

A. Assign P[3] the value 1.
B. Assign PBI[3] the value bj.

iii. Otherwise, assign P[3] the value zero.

(c) Otherwise, assign P[3] the value zero.

10. Otherwise, assign P[3] the value zero.

11. If none of the values P[0], P[1], P[2], nor P[3] are non-zero, then assign **DCPRED** the value **LASTDC**[rfi].

12. Otherwise:

(a) Assign the array W and the variable PDIV the values from the row of Table 7.47 corresonding to the values of each P[i].

P[0] (L)	P[1] (DL)	P[2] (D)	P[3] (DR)	W[3] (L)	W[1] (DL)	W[2] (D)	W[3] (DR)	PDIV
1	0	0	0	1	0	0	0	1
0	1	0	0	0	1	0	0	1
1	1	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	2
0	1	1	0	0	0	1	0	1
1	1	1	0	29	−26	29	0	32
0	0	0	1	0	0	0	1	1
1	0	0	1	75	0	0	53	128
0	1	0	1	0	1	0	1	2
1	1	0	1	75	0	0	53	128
0	0	1	1	0	0	1	0	1
1	0	1	1	75	0	0	53	128
0	1	1	1	0	3	10	3	16
1	1	1	1	29	−26	29	0	32

Table 7.47: Weights and Divisors for Each Set of Available DC Predictors

(b) Assign **DCPRED** the value zero.

(c) If P[0] is non-zero, assign **DCPRED** the value (**DCPRED** + W[0] ∗ **COEFFS**[PBI[0]][0]).

(d) If P[1] is non-zero, assign **DCPRED** the value (**DCPRED** + W[1] ∗ **COEFFS**[PBI[1]][0]).

(e) If P[2] is non-zero, assign **DCPRED** the value (**DCPRED** + W[2] ∗ **COEFFS**[PBI[2]][0]).

(f) If P[3] is non-zero, assign **DCPRED** the value (**DCPRED** + W[3] ∗ **COEFFS**[PBI[3]][0]).

(g) Assign **DCPRED** the value (**DCPRED**//PDIV).

(h) If P[0], P[1], and P[2] are all non-zero:

i. If |**DCPRED** − **COEFFS**[PBI[2]][0]| is greater than 128, assign **DCPRED** the value **COEFFS**[PBI[2]][0].

ii. Otherwise, if |**DCPRED** − **COEFFS**[PBI[0]][0]| is greater than 128, assign **DCPRED** the value **COEFFS**[PBI[0]][0].

iii. Otherwise, if |**DCPRED** − **COEFFS**[PBI[1]][0]| is greater than 128, assign **DCPRED** the value **COEFFS**[PBI[1]][0].

7.8.2 Inverting the DC Prediction Process

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.
MBMODES	Integer Array	3	No	An **NMBS**-element array of coding modes for each macro block.
COEFFS	2D Integer Array	16	Yes	An **NBS**$\times 64$ array of quantized DCT coefficient values for each block in zig-zag order.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
COEFFS	2D Integer Array	16	Yes	An **NBS**$\times 64$ array of quantized DCT coefficient values for each block in zig-zag order. The DC value of each block will be updated.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
LASTDC	Integer Array	16	Yes	A 3-element array containing the most recently decoded DC value, one for inter mode and for each reference frame.
DCPRED	Integer	11	Yes	The predicted DC value for the current block.
DC	Integer	17	Yes	The actual DC value for the current block.
bi	Integer	36	No	The index of the current block in coded order.
mbi	Integer	32	No	The index of the macro block containing block *bi*.
rfi	Integer	2	No	The index of the reference frame indicated by the coding mode for macro block *mbi*.

Name	Type	Size (bits)	Signed?	Description and restrictions
pli	Integer	2	No	A color plane index.

This procedure describes the complete process of undoing the DC prediction to recover the original DC values. Because it is possible to add a value as large as 580 to the predicted DC coefficient value at every block, which will then be used to increase the predictor for the next block, the reconstructed DC value could overflow a 16-bit integer. This is handled by truncating the result to a 16-bit signed representation, simply throwing away any higher bits in the two's complement representation of the number.

1. For each consecutive value of *pli* from 0 to 2:
   (a) Assign LASTDC[0] the value zero.
   (b) Assign LASTDC[1] the value zero.
   (c) Assign LASTDC[2] the value zero.
   (d) For each block of color plane *pli* in *raster* order, with coded-order index *bi*:
      i. If **BCODED**[*bi*] is non-zero:
         A. Compute the value DCPRED using the procedure outlined in Section 7.8.1.
         B. Assign DC the value (**COEFFS**[*bi*][0] + DCPRED).
         C. Truncate DC to a 16-bit representation by dropping any higher-order bits.
         D. Assign **COEFFS**[*bi*][0] the value DC.
         E. Assign *mbi* the index of the macro block containing block *bi*.
         F. Assign *rfi* the value of the Reference Frame Index column of Table 7.46 corresponding to **MBMODES**[*mbi*].
         G. Assign LASTDC[*rfi*] the value DC.

7.9 Reconstruction

At this stage, the complete contents of the data packet have been decoded. All that remains is to reconstruct the contents of the new frame. This is applied on a block by block basis, and as each block is independent, the order they are processed in does not matter.

7.9.1 Predictors

For each block, a predictor is formed based on its coding mode and motion vector. There are three basic types of predictors: the intra predictor, the whole-pixel predictor, and the half-pixel predictor. The former is used for all blocks

coded in INTRA mode, while all other blocks use one of the latter two. The whole-pixel predictor is used if the fractional part of both motion vector components is zero, otherwise the half-pixel predictor is used.

The Intra Predictor

Input parameters: None.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
PRED	2D Integer Array	8	No	An 8×8 array of predictor values to use for INTRA coded blocks.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
bx	Integer	3	No	The horizontal pixel index in the block.
by	Integer	3	No	The vertical pixel index in the block.

The intra predictor is nothing more than the constant value 128. This is applied for the sole purpose of centering the range of possible DC values for INTRA blocks around zero.

1. For each value of *by* from 0 to 7, inclusive:
   (a) For each value of *bx* from 0 to 7, inclusive:
      i. Assign **PRED**[*by*][*bx*] the value 128.

The Whole-Pixel Predictor

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RPW	Integer	20	No	The width of the current plane of the reference frame in pixels.
RPH	Integer	20	No	The height of the current plane of the reference frame in pixels.

Name	Type	Size (bits)	Signed?	Description and restrictions
REFP	2D Integer Array	8	No	A **RPH** $\times$ **RPW** array containing the contents of the current plane of the reference frame.
BX	Integer	20	No	The horizontal pixel index of the lower-left corner of the current block.
BY	Integer	20	No	The vertical pixel index of the lower-left corner of the current block.
MVX	Integer	5	No	The horizontal component of the block motion vector. This is always a whole-pixel value.
MVY	Integer	5	No	The vertical component of the block motion vector. This is always a whole-pixel value.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
PRED	2D Integer Array	8	No	An 8×8 array of predictor values to use for INTER coded blocks.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
bx	Integer	3	Yes	The horizontal pixel index in the block.
by	Integer	3	Yes	The vertical pixel index in the block.
rx	Integer	20	No	The horizontal pixel index in the reference frame.
ry	Integer	20	No	The vertical pixel index in the reference frame.

The whole pixel predictor simply copies verbatim the contents of the reference frame pointed to by the block's motion vector. If the vector points outside the reference frame, then the closest value on the edge of the reference frame is used instead. In practice, this is usually implemented by expanding the size

of the reference frame by 8 or 16 pixels on each side—depending on whether or not the corresponding axis is subsampled in the current plane—and copying the border pixels into this region.

1. For each value of by from 0 to 7, inclusive:
   (a) Assign ry the value $(\mathbf{BY} + \mathbf{MVY} + by)$.
   (b) If ry is greater than $(\mathbf{RPH} - 1)$, assign ry the value $(\mathbf{RPH} - 1)$.
   (c) If ry is less than zero, assign ry the value zero.
   (d) For each value of bx from 0 to 7, inclusive:
      i. Assign rx the value $(\mathbf{BX} + \mathbf{MVX} + bx)$.
      ii. If rx is greater than $(\mathbf{RPW} - 1)$, assign rx the value $(\mathbf{RPW} - 1)$.
      iii. If rx is less than zero, assign rx the value zero.
      iv. Assign $\mathbf{PRED}[by][bx]$ the value $\mathbf{REFP}[ry][rx]$.

The Half-Pixel Predictor

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RPW	Integer	20	No	The width of the current plane of the reference frame in pixels.
RPH	Integer	20	No	The height of the current plane of the reference frame in pixels.
REFP	2D Integer Array	8	No	A $\mathbf{RPH} \times \mathbf{RPW}$ array containing the contents of the current plane of the reference frame.
BX	Integer	20	No	The horizontal pixel index of the lower-left corner of the current block.
BY	Integer	20	No	The vertical pixel index of the lower-left corner of the current block.
MVX	Integer	5	No	The horizontal component of the first whole-pixel motion vector.
MVY	Integer	5	No	The vertical component of the first whole-pixel motion vector.
MVX2	Integer	5	No	The horizontal component of the second whole-pixel motion vector.
MVY2	Integer	5	No	The vertical component of the second whole-pixel motion vector.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
PRED	2D Integer Array	8	No	An 8×8 array of predictor values to use for INTER coded blocks.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
bx	Integer	3	Yes	The horizontal pixel index in the block.
by	Integer	3	Yes	The vertical pixel index in the block.
rx1	Integer	20	No	The first horizontal pixel index in the reference frame.
ry1	Integer	20	No	The first vertical pixel index in the reference frame.
rx2	Integer	20	No	The second horizontal pixel index in the reference frame.
ry2	Integer	20	No	The second vertical pixel index in the reference frame.

If one or both of the components of the block motion vector is not a whole-pixel value, then the half-pixel predictor is used. The half-pixel predictor converts the fractional motion vector into two whole-pixel motion vectors. The first is formed by truncating the values of each component towards zero, and the second is formed by truncating them away from zero. The contributions from the reference frame at the locations pointed to by each vector are averaged, truncating towards negative infinity.

Only two samples from the reference frame contribute to each predictor value, even if both components of the motion vector have non-zero fractional components. Motion vector components with quarter-pixel accuracy in the chroma planes are treated exactly the same as those with half-pixel accuracy. Any non-zero fractional part gets rounded one way in the first vector, and the other way in the second.

1. For each value of *by* from 0 to 7, inclusive:
   (a) Assign *ry1* the value (**BY** + **MVY1** + *by*).
   (b) If *ry1* is greater than (**RPH** − 1), assign *ry1* the value (**RPH** − 1).
   (c) If *ry1* is less than zero, assign *ry1* the value zero.

(d) Assign *ry2* the value (**BY** + **MVY2** + *by*).

(e) If *ry2* is greater than (**RPH** − 1), assign *ry2* the value (**RPH** − 1).

(f) If *ry2* is less than zero, assign *ry2* the value zero.

(g) For each value of *bx* from 0 to 7, inclusive:

i. Assign *rx1* the value (**BX** + **MVX1** + *bx*).

ii. If *rx1* is greater than (**RPW** − 1), assign *rx1* the value (**RPW** − 1).

iii. If *rx1* is less than zero, assign *rx1* the value zero.

iv. Assign *rx2* the value (**BX** + **MVX2** + *bx*).

v. If *rx2* is greater than (**RPW** − 1), assign *rx2* the value (**RPW** − 1).

vi. If *rx2* is less than zero, assign *rx2* the value zero.

vii. Assign **PRED**[*by*][*bx*] the value

$$(\mathbf{REFP}[ry1][rx1] + \mathbf{REFP}[ry2][rx2]) >> 1.$$

7.9.2 Dequantization

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
COEFFS	2D Integer Array	16	Yes	An **NBS** × 64 array of quantized DCT coefficient values for each block in zig-zag order.
ACSCALE	Integer array	16	No	A 64-element array of scale values for AC coefficients for each *qi* value.
DCSCALE	Integer array	16	No	A 64-element array of scale values for the DC coefficient for each *qi* value.
BMS	2D Integer array	8	No	A **NBMS** × 64 array containing the base matrices.
NQRS	2D Integer array	6	No	A 2 × 3 array containing the number of quant ranges for a given *qti* and *pli*, respectively. This is at most 63.
QRSIZES	3D Integer array	6	No	A 2 × 3 × 63 array of the sizes of each quant range for a given *qti* and *pli*, respectively. Only the first **NQRS**[*qti*][*pli*] values are used.

Name	Type	Size (bits)	Signed?	Description and restrictions
QRBMIS	3D Integer array	9	No	A $2 \times 3 \times 64$ array of the *bmi*'s used for each quant range for a given *qti* and *pli*, respectively. Only the first (**NQRS**[*qti*][*pli*] + 1) values are used.
qti	Integer	1	No	A quantization type index. See Table 3.1.
pli	Integer	2	No	A color plane index. See Table 2.1.
qi0	Integer	6	No	The quantization index of the DC coefficient.
qi	Integer	6	No	The quantization index of the AC coefficients.
bi	Integer	36	No	The index of the current block in coded order.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
DQC	Integer Array	14	Yes	A 64-element array of dequantized DCT coefficients in natural order (cf. Section 2.6).

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
QMAT	Integer array	16	No	A 64-element array of quantization values for each DCT coefficient in natural order.
ci	Integer	6	No	The DCT coefficient index in natural order.
zzi	Integer	6	No	The DCT coefficient index in zig-zag order.
C	Integer	29	Yes	A single dequantized coefficient.

This procedure takes the quantized DCT coefficient values in zig-zag order for a single block—after DC prediction has been undone—and returns the dequantized values in natural order. If large coefficient values are decoded for coarsely quantized coefficients, the resulting dequantized value can be significantly larger than 16 bits. Such a coefficient is truncated to a signed 16-bit representation by discarding the higher-order bits of its twos-complement representation.

Although this procedure recomputes the quantization matrices from the parameters in the setup header for each block, there are at most six different ones used for each color plane. An efficient implementation could compute them once in advance.

1. Using **ACSCALE**, **DCSCALE**, **BMS**, **NQRS**, **QRSIZES**, **QRBMIS**, ***qti***, ***pli***, and ***qi0***, use the procedure given in Section 6.4.3 to compute the DC quantization matrix QMAT.
2. Assign C the value **COEFFS**[***bi***][0] $*$ QMAT[0].
3. Truncate C to a 16-bit representation by dropping any higher-order bits.
4. Assign **DQC**[0] the value C.
5. Using **ACSCALE**, **DCSCALE**, **BMS**, **NQRS**, **QRSIZES**, **QRBMIS**, ***qti***, ***pli***, and ***qi***, use the procedure given in Section 6.4.3 to compute the AC quantization matrix QMAT.
6. For each value of ci from 1 to 63, inclusive:
   (a) Assign zzi the index in zig-zag order corresponding to ci. E.g., the value at row ($ci//8$) and column ($ci\%8$) in Figure 2.8
   (b) Assign C the value **COEFFS**[***bi***][zzi] $*$ QMAT[ci].
   (c) Truncate C to a 16-bit representation by dropping any higher-order bits.
   (d) Assign **DQC**[ci] the value C.

7.9.3 The Inverse DCT

The 2D inverse DCT is separated into two applications of the 1D inverse DCT. The transform is first applied to each row, and then applied to each column of the result.

Each application of the 1D inverse DCT scales the values by a factor of two relative to the orthonormal version of the transform, for a total scale factor of four for the 2D transform. It is assumed that a similar scale factor is applied during the forward DCT used in the encoder, so that a division by 16 is required after the transform has been applied in both directions. The inclusion of this scale factor allows the integerized transform to operate with increased precision. All divisions throughout the transform are implemented with right shifts. Only the final division by 16 is rounded, with ties rounded towards positive infinity.

All intermediate values are truncated to a 32-bit signed representation by discarding any higher-order bits in their two's complement representation. The final output of each 1D transform is truncated to a 16-bit signed value in the same manner. In practice, if the high word of a 16×16 bit multiplication can be obtained directly, 16 bits is sufficient for every calculation except scaling by $C4$. Thus we truncate to 16 bits before that multiplication to allow an implementation entirely in 16-bit registers. Implementations using larger registers must sign-extend the 16-bit value to maintain compatibility.

Note that if 16-bit register are used, overflow in the additions and subtractions should be handled using *unsaturated* arithmetic. That is, the high-order bits should be discarded and the low-order bits retained, instead of clamping the result to the maximum or minimum value. This allows the maximum flexibility in re-ordering these instructions without deviating from this specification.

The 1D transform can only overflow if input coefficients larger than ± 6201 are present. However, the result of applying the 2D forward transform on pixel values in the range $-255 \ldots 255$ can be as large as ± 8157 due to the scale factor of four that is applied, and quantization errors could make this even larger. Therefore, the coefficients cannot simply be clamped into a valid range before the transform.

The 1D Inverse DCT

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
Y	Integer Array	16	Yes	An 8-element array of DCT coefficients.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
X	Integer Array	16	Yes	An 8-element array of output values.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
T	Integer Array	32	Yes	An 8-element array containing the current value of each signal line.
R	Integer	32	Yes	A temporary value.

A compliant decoder MUST use the exact implementation of the inverse DCT defined in this specification. Some operations may be re-ordered, but the result must be precisely equivalent. This is a design decision that limits some avenues of decoder optimization, but prevents any drift in the prediction loop. Theora uses a 16-bit integerized approximation of of the 8-point 1D inverse DCT based on the Chen factorization [CSF77]. It requires 16 multiplications and 26 additions and subtractions.

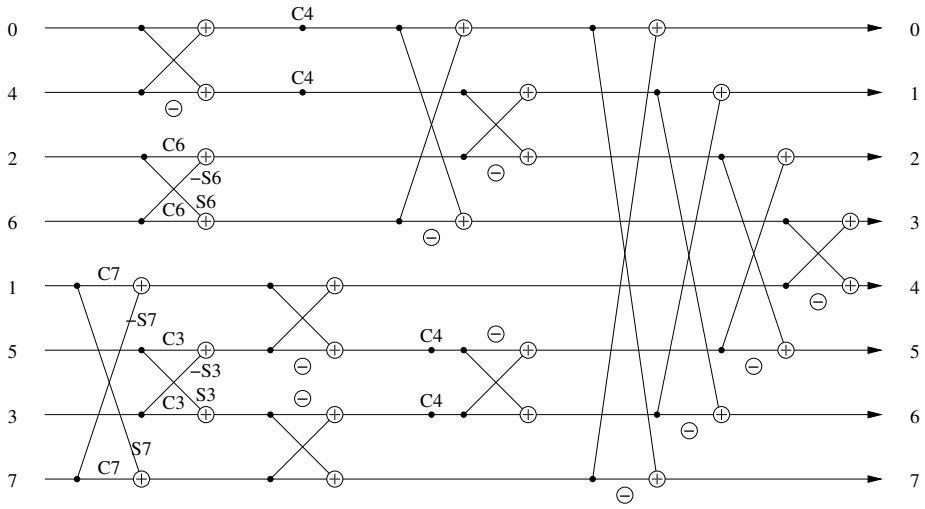


Figure 7.1: Signal Flow Graph for the 1D Inverse DCT

A signal flow graph of the transformation is presented in Figure 7.1. This graph provides a good visualization of which parts of the transform are parallelizable. Time increases from left to right.

Each signal line is involved in an operation where the line is marked with a dot $\cdot$ or a circled plus sign $\oplus$. The constants Ci and Sj are the 16-bit integer approximations of $\cos(\frac{i\pi}{16})$ and $\sin(\frac{j\pi}{16})$ listed in Table 7.65. When they appear next to a signal line, the value on that line is scaled by the given constant. A circled minus sign $\ominus$ next to a signal line indicates that the value on that line is negated.

Operations on a single signal path through the graph cannot be reordered, but operations on different paths may be, or may be executed in parallel. Dif-

ferent graphs may be obtainable using the associative, commutative, and distributive properties of unsaturated arithmetic. The column of numbers on the left represents an initial permutation of the input DCT coefficients. The column on the right represents the unpermuted output. One can be obtained by bit-reversing the 3-bit binary representation of the other.

Ci	Sj	Value
C1	S7	64277
C2	S6	60547
C3	S5	54491
C4	S4	46341
C5	S3	36410
C6	S2	25080
C7	S1	12785

Table 7.65: 16-bit Approximations of Sines and Cosines

1. Assign T[0] the value $\mathbf{Y}[0] + \mathbf{Y}[4]$.
2. Truncate T[0] to a 16-bit signed representation by dropping any higher-order bits.
3. Assign T[0] the value $\mathrm{C4} * \mathrm{T}[0] >> 16$.
4. Assign T[1] the value $\mathbf{Y}[0] - \mathbf{Y}[4]$.
5. Truncate T[1] to a 16-bit signed representation by dropping any higher-order bits.
6. Assign T[1] the value $\mathrm{C4} * \mathrm{T}[1] >> 16$.
7. Assign T[2] the value $(\mathrm{C6} * \mathbf{Y}[2] >> 16) - (\mathrm{S6} * \mathbf{Y}[6] >> 16)$.
8. Assign T[3] the value $(\mathrm{S6} * \mathbf{Y}[2] >> 16) + (\mathrm{C6} * \mathbf{Y}[6] >> 16)$.
9. Assign T[4] the value $(\mathrm{C7} * \mathbf{Y}[1] >> 16) - (\mathrm{S7} * \mathbf{Y}[7] >> 16)$.
10. Assign T[5] the value $(\mathrm{C3} * \mathbf{Y}[5] >> 16) - (\mathrm{S3} * \mathbf{Y}[3] >> 16)$.
11. Assign T[6] the value $(\mathrm{S3} * \mathbf{Y}[5] >> 16) + (\mathrm{C3} * \mathbf{Y}[3] >> 16)$.
12. Assign T[7] the value $(\mathrm{S7} * \mathbf{Y}[1] >> 16) + (\mathrm{C7} * \mathbf{Y}[7] >> 16)$.
13. Assign R the value $\mathrm{T}[4] + \mathrm{T}[5]$.
14. Assign T[5] the value $\mathrm{T}[4] - \mathrm{T}[5]$.
15. Truncate T[5] to a 16-bit signed representation by dropping any higher-order bits.

16. Assign T[5] the value $C4 * T[5] >> 16$.
17. Assign T[4] the value R.
18. Assign R the value $T[7] + T[6]$.
19. Assign T[6] the value $T[7] - T[6]$.
20. Truncate T[6] to a 16-bit signed representation by dropping any higher-order bits.
21. Assign T[6] the value $C4 * T[6] >> 16$.
22. Assign T[7] the value R.
23. Assign R the value $T[0] + T[3]$.
24. Assign T[3] the value $T[0] - T[3]$.
25. Assign T[0] the value R.
26. Assign R the value $T[1] + T[2]$
27. Assign T[2] the value $T[1] - T[2]$
28. Assign T[1] the value R.
29. Assign R the value $T[6] + T[5]$.
30. Assign T[5] the value $T[6] - T[5]$.
31. Assign T[6] the value R.
32. Assign R the value $T[0] + T[7]$.
33. Truncate R to a 16-bit signed representation by dropping any higher-order bits.
34. Assign $\mathbf{X}[0]$ the value R.
35. Assign R the value $T[1] + T[6]$.
36. Truncate R to a 16-bit signed representation by dropping any higher-order bits.
37. Assign $\mathbf{X}[1]$ the value R.
38. Assign R the value $T[2] + T[5]$.
39. Truncate R to a 16-bit signed representation by dropping any higher-order bits.
40. Assign $\mathbf{X}[2]$ the value R.
41. Assign R the value $T[3] + T[4]$.

42. Truncate R to a 16-bit signed representation by dropping any higher-order bits.
43. Assign **X**[3] the value R.
44. Assign R the value T[3] − T[4].
45. Truncate R to a 16-bit signed representation by dropping any higher-order bits.
46. Assign **X**[4] the value R.
47. Assign R the value T[2] − T[5].
48. Truncate R to a 16-bit signed representation by dropping any higher-order bits.
49. Assign **X**[5] the value R.
50. Assign R the value T[1] − T[6].
51. Truncate R to a 16-bit signed representation by dropping any higher-order bits.
52. Assign **X**[6] the value R.
53. Assign R the value T[0] − T[7].
54. Truncate R to a 16-bit signed representation by dropping any higher-order bits.
55. Assign **X**[7] the value R.

The 2D Inverse DCT

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
DQC	Integer Array	14	Yes	A 64-element array of dequantized DCT coefficients in natural order (cf. Section 2.6).

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RES	2D Integer Array	16	Yes	An 8×8 array containing the decoded residual for the current block.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
ci	Integer	3	No	The column index.
ri	Integer	3	No	The row index.
Y	Integer Array	16	Yes	An 8-element array of 1D iDCT input values.
X	Integer Array	16	Yes	An 8-element array of 1D iDCT output values.

This procedure applies the 1D inverse DCT transform 16 times to a block of dequantized coefficients: once for each of the 8 rows, and once for each of the 8 columns of the result. Note that the coordinate system used for the columns is the same right-handed coordinate system used by the rest of Theora. Thus, the column is indexed from bottom to top, not top to bottom. The final values are divided by sixteen, rounding with ties rounded towards postive infinity.

1. For each value of ri from 0 to 7:
   (a) For each value of ci from 0 to 7:
      i. Assign Y[ci] the value **DQC**[$ri * 8 + ci$].
   (b) Compute X, the 1D inverse DCT of Y using the procedure described in Section 7.9.3.
   (c) For each value of ci from 0 to 7:
      i. Assign **RES**[ri][ci] the value X[ci].
2. For each value of ci from 0 to 7:
   (a) For each value of ri from 0 to 7:
      i. Assign Y[ri] the value **RES**[ri][ci].
   (b) Compute X, the 1D inverse DCT of Y using the procedure described in Section 7.9.3.
   (c) For each value of ri from 0 to 7:
      i. Assign **RES**[ri][ci] the value (X[ri] + 8) >> 4.

The 1D Forward DCT (Non-Normative)

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
X	Integer Array	14	Yes	An 8-element array of input values.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
Y	Integer Array	16	Yes	An 8-element array of DCT coefficients.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
T	Integer Array	16	Yes	An 8-element array containing the current value of each signal line.
R	Integer	16	Yes	A temporary value.

The forward transform used in the encoder is not mandated by this standard as the inverse one is. Precise equivalence in the inverse transform alone is all that is required to guarantee that there is no mismatch in the prediction loop between encoder and any compliant decoder implementation. However, a forward transform is provided here as a convenience for implementing an encoder. This is the version of the transform used by Xiph.org's Theora encoder, which is the same as that used by VP3. Like the inverse DCT, it is first applied to each row, and then applied to each column of the result.

The signal flow graph for the forward transform is given in Figure 7.2. It is largely the reverse of the flow graph given for the inverse DCT. It is important to note that the signs on the constants in the rotations have changed, and the C4 scale factors on one of the lower butterflies now appear on the opposite side. The column of numbers on the left represents the unpermuted input, and the column on the right the permuted output DCT coefficients.

A proper division by 2^{16} is done after the multiplications instead of a shift in the forward transform. This can be implemented quickly by adding an offset

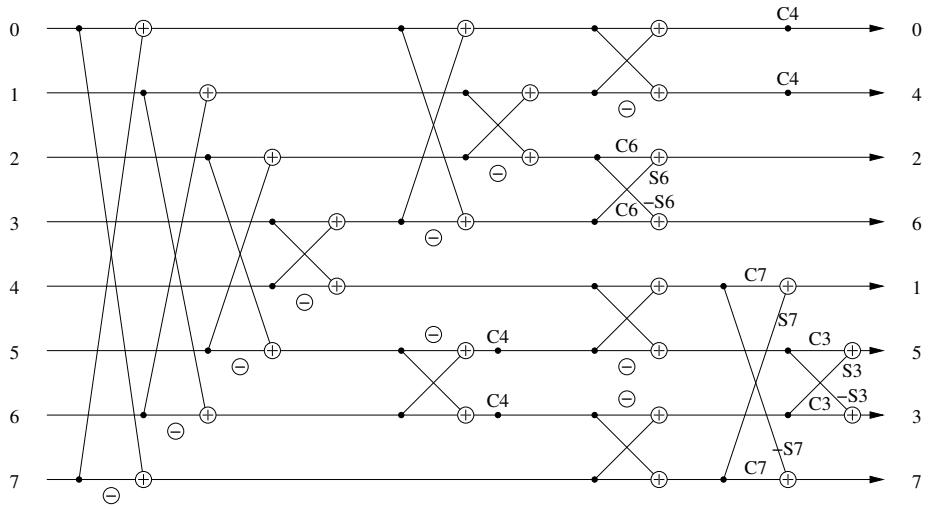


Figure 7.2: Signal Flow Graph for the 1D Forward DCT

of `0xFFFF` if the number is negative, and then shifting as before. This slightly increases the computational complexity of the transform. Unlike the inverse DCT, 16-bit registers and a $16 \times 16 \rightarrow 32$ bit multiply are sufficient to avoid any overflow, so long as the input is in the range $-6270 \ldots 6270$, which is larger than required.

1. Assign T[0] the value $\mathbf{X}[0] + \mathbf{X}[7]$.
2. Assign T[1] the value $\mathbf{X}[1] + \mathbf{X}[6]$.
3. Assign T[2] the value $\mathbf{X}[2] + \mathbf{X}[5]$.
4. Assign T[3] the value $\mathbf{X}[3] + \mathbf{X}[4]$.
5. Assign T[4] the value $\mathbf{X}[3] - \mathbf{X}[4]$.
6. Assign T[5] the value $\mathbf{X}[2] - \mathbf{X}[5]$.
7. Assign T[6] the value $\mathbf{X}[1] - \mathbf{X}[6]$.
8. Assign T[7] the value $\mathbf{X}[0] - \mathbf{X}[7]$.
9. Assign R the value T[0] + T[3].
10. Assign T[3] the value T[0] − T[3].
11. Assign T[0] the value R.
12. Assign R the value T[1] + T[2].

13. Assign T[2] the value T[1] − T[2].
14. Assign T[1] the value R.
15. Assign R the value T[6] − T[5].
16. Assign T[6] the value (C4 ∗ (T[6] + T[5]))//16.
17. Assign T[5] the value (C4 ∗ R)//16.
18. Assign R the value T[4] + T[5].
19. Assign T[5] the value T[4] − T[5].
20. Assign T[4] the value R.
21. Assign R the value T[7] + T[6].
22. Assign T[6] the value T[7] − T[6].
23. Assign T[7] the value R.
24. Assign **Y**[0] the value (C4 ∗ (T[0] + T[1]))//16.
25. Assign **Y**[4] the value (C4 ∗ (T[0] − T[1]))//16.
26. Assign **Y**[2] the value ((S6 ∗ T[3])//16) + ((C6 ∗ T[2])//16).
27. Assign **Y**[6] the value ((C6 ∗ T[3])//16) − ((S6 ∗ T[2])//16).
28. Assign **Y**[1] the value ((S7 ∗ T[7])//16) + ((C7 ∗ T[4])//16).
29. Assign **Y**[5] the value ((S3 ∗ T[6])//16) + ((C3 ∗ T[5])//16).
30. Assign **Y**[3] the value ((C3 ∗ T[6])//16) − ((S3 ∗ T[5])//16).
31. Assign **Y**[7] the value ((C7 ∗ T[7])//16) − ((S7 ∗ T[4])//16).

7.9.4 The Complete Reconstruction Algorithm

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
ACSCALE	Integer array	16	No	A 64-element array of scale values for AC coefficients for each *qi* value.
DCSCALE	Integer array	16	No	A 64-element array of scale values for the DC coefficient for each *qi* value.

Name	Type	Size (bits)	Signed?	Description and restrictions
BMS	2D Integer array	8	No	A **NBMS** $\times 64$ array containing the base matrices.
NQRS	2D Integer array	6	No	A 2×3 array containing the number of quant ranges for a given *qti* and *pli*, respectively. This is at most 63.
QRSIZES	3D Integer array	6	No	A $2 \times 3 \times 63$ array of the sizes of each quant range for a given *qti* and *pli*, respectively. Only the first **NQRS**$[qti][pli]$ values are used.
QRBMIS	3D Integer array	9	No	A $2 \times 3 \times 64$ array of the *bmi*'s used for each quant range for a given *qti* and *pli*, respectively. Only the first (**NQRS**$[qti][pli]+1$) values are used.
RPYW	Integer	20	No	The width of the Y' plane of the reference frames in pixels.
RPYH	Integer	20	No	The height of the Y' plane of the reference frames in pixels.
RPCW	Integer	20	No	The width of the C_b and C_r planes of the reference frames in pixels.
RPCH	Integer	20	No	The height of the C_b and C_r planes of the reference frames in pixels.
GOLDREFY	2D Integer Array	8	No	A **RPYH** $\times$ **RPYW** array containing the contents of the Y' plane of the golden reference frame.
GOLDREFCB	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_b plane of the golden reference frame.

Name	Type	Size (bits)	Signed?	Description and restrictions
GOLDREFCR	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_r plane of the golden reference frame.
PREVREFY	2D Integer Array	8	No	A **RPYH** $\times$ **RPYW** array containing the contents of the Y' plane of the previous reference frame.
PREVREFCB	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_b plane of the previous reference frame.
PREVREFCR	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_r plane of the previous reference frame.
NBS	Integer	36	No	The total number of blocks in a frame.
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.
MBMODES	Integer Array	3	No	An **NMBS**-element array of coding modes for each macro block.
MVECTS	Array of 2D Integer Vectors	6	Yes	An **NBS**-element array of motion vectors for each block.
COEFFS	2D Integer Array	16	Yes	An **NBS** $\times$ 64 array of quantized DCT coefficient values for each block in zig-zag order.
NCOEFFS	Integer Array	7	No	An **NBS**-element array of the coefficient count for each block.
QIS	Integer array	6	No	An **NQIS**-element array of qi values.
QIIS	Integer Array	2	No	An **NBS**-element array of qii values for each block.

Name	Type	Size (bits)	Signed?	Description and restrictions

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RECY	2D Integer Array	8	No	A **RPYH** $\times$ **RPYW** array containing the contents of the Y' plane of the reconstructed frame.
RECCB	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_b plane of the reconstructed frame.
RECCR	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_r plane of the reconstructed frame.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
RPW	Integer	20	No	The width of the current plane of the current reference frame in pixels.
RPH	Integer	20	No	The height of the current plane of the current reference frame in pixels.
REFP	2D Integer Array	8	No	A **RPH** $\times$ **RPW** array containing the contents of the current plane of the current reference frame.
BX	Integer	20	No	The horizontal pixel index of the lower-left corner of the current block.
BY	Integer	20	No	The vertical pixel index of the lower-left corner of the current block.
MVX	Integer	5	No	The horizontal component of the first whole-pixel motion vector.
MVY	Integer	5	No	The vertical component of the first whole-pixel motion vector.
MVX2	Integer	5	No	The horizontal component of the second whole-pixel motion vector.

Name	Type	Size (bits)	Signed?	Description and restrictions
MVY2	Integer	5	No	The vertical component of the second whole-pixel motion vector.
PRED	2D Integer Array	8	No	An 8×8 array of predictor values to use for the current block.
RES	2D Integer Array	16	Yes	An 8×8 array containing the decoded residual for the current block.
QMAT	Integer array	16	No	A 64-element array of quantization values for each DCT coefficient in natural order.
DC	Integer	29	Yes	The dequantized DC coefficient of a block.
P	Integer	17	Yes	A reconstructed pixel value.
bi	Integer	36	No	The index of the current block in coded order.
mbi	Integer	32	No	The index of the macro block containing block *bi*.
pli	Integer	2	No	The color plane index of the current block.
rfi	Integer	2	No	The index of the reference frame indicated by the coding mode for macro block *mbi*.
bx	Integer	3	No	The horizontal pixel index in the block.
by	Integer	3	No	The vertical pixel index in the block.
qti	Integer	1	No	A quantization type index. See Table 3.1.
qi0	Integer	6	No	The quantization index of the DC coefficient.
qi	Integer	6	No	The quantization index of the AC coefficients.

This section takes the decoded packet data and uses the previously defined procedures to reconstruct each block of the current frame. For coded blocks, a predictor is formed using the coding mode and, if applicable, the motion vector, and then the residual is computed from the quantized DCT coefficients. For uncoded blocks, the contents of the co-located block are copied from the previous frame and the residual is cleared to zero. Then the predictor and residual are added, and the result clamped to the range $0 \ldots 255$ and stored in the current frame.

In the special case that a block contains only a DC coefficient, the dequanti-

zation and inverse DCT transform is skipped. Instead the constant pixel value for the entire block is computed in one step. Note that the truncation of intermediate operations is omitted and the final rounding is slightly different in this case. The check for whether or not the block contains only a DC coefficient is based on the coefficient count returned from the token decode procedure of Section 7.7, and not by checking to see if the remaining coefficient values are zero. Also note that even when the coefficient count indicates the block contains zero coefficients, the DC coefficient is still processed, as undoing DC prediction might have made it non-zero.

After this procedure, the frame is completely reconstructed, but before it can be used as a reference frame, a loop filter must be run over it to help reduce blocking artifacts. This is detailed in Section 7.10.

1. Assign *qi0* the value **QIS**[0].
2. For each value of *bi* from 0 to (**NBS** − 1):
   (a) Assign *pli* the index of the color plane block *bi* belongs to.
   (b) Assign BX the horizontal pixel index of the lower-left corner of block *bi*.
   (c) Assign BY the vertical pixel index of the lower-left corner of block *bi*.
   (d) If **BCODED**[*bi*] is non-zero:
      i. Assign *mbi* the index of the macro block containing block *bi*.
      ii. If **MBMODES**[*mbi*] is 1 (INTRA), assign *qti* the value 0.
      iii. Otherwise, assign *qti* the value 1.
      iv. Assign *rfi* the value of the Reference Frame Index column of Table 7.46 corresponding to **MBMODES**[*mbi*].
      v. If *rfi* is zero, compute PRED using the procedure given in Section 7.9.1.
      vi. Otherwise:
         A. Assign REFP, RPW, and RPH the values given in Table 7.75 corresponding to current value of *rfi* and *pli*.

rfi	*pli*	REFP	RPW	RPH
1	0	**PREVREFY**	**RPYW**	**RPYH**
1	1	**PREVREFCB**	**RPCW**	**RPCH**
1	2	**PREVREFCR**	**RPCW**	**RPCH**
2	0	**GOLDREFY**	**RPYW**	**RPYH**
2	1	**GOLDREFCB**	**RPCW**	**RPCH**
2	2	**GOLDREFCR**	**RPCW**	**RPCH**

Table 7.75: Reference Planes and Sizes for Each *rfi* and *pli*

B. Assign MVX the value

$$\lfloor |\mathbf{MVECTS}[bi]_x| \rfloor * \text{sign}(\mathbf{MVECTS}[bi]_x).$$

C. Assign MVY the value

$$\lfloor |\mathbf{MVECTS}[bi]_y| \rfloor * \text{sign}(\mathbf{MVECTS}[bi]_y).$$

D. Assign MVX2 the value

$$\lceil |\mathbf{MVECTS}[bi]_x| \rceil * \text{sign}(\mathbf{MVECTS}[bi]_x).$$

E. Assign MVY2 the value

$$\lceil |\mathbf{MVECTS}[bi]_y| \rceil * \text{sign}(\mathbf{MVECTS}[bi]_y).$$

F. If MVX equals MVX2 and MVY equals MVY2, use the values REFP, RPW, RPH, BX, BY, MVX, and MVY, compute PRED using the procedure given in Section 7.9.1.

G. Otherwise, use the values REFP, RPW, RPH, BX, BY, MVX, MVY, MVX2, and MVY2 to compute PRED using the procedure given in Section 7.9.1.

vii. If **NCOEFFS**[*bi*] is less than 2:

A. Using **ACSCALE**, **DCSCALE**, **BMS**, **NQRS**, **QRSIZES**, **QRBMIS**, *qti*, *pli*, and *qi0*, use the procedure given in Section 6.4.3 to compute the DC quantization matrix QMAT.

B. Assign DC the value

$$(\mathbf{COEFFS}[\boldsymbol{bi}][0] * \text{QMAT}[0] + 15) >> 5.$$

C. Truncate DC to a 16-bit signed representation by dropping any higher-order bits.

D. For each value of *by* from 0 to 7, and each value of *bx* from 0 to 7, assign RES[*by*][*bx*] the value DC.

viii. Otherwise:

A. Assign *qi* the value **QIS**[**QIIS**[*bi*]].

B. Using **ACSCALE**, **DCSCALE**, **BMS**, **NQRS**, **QRSIZES**, **QRBMIS**, *qti*, *pli*, *qi0*, and *qi*, compute DQC using the procedure given in Section 7.9.2.

C. Using DQC, compute RES using the procedure given in Section 7.9.3.

(e) Otherwise:

i. Assign *rfi* the value 1.

ii. Assign REFP, RPW, and RPH the values given in Table 7.75 corresponding to current value of *rfi* and *pli*.

iii. Assign MVX the value 0.

iv. Assign MVY the value 0.

v. Using the values REFP, RPW, RPH, BX, BY, MVX, and MVY, compute PRED using the procedure given in Section 7.9.1. This is simply a copy of the co-located block in the previous reference frame.

vi. For each value of *by* from 0 to 7, and each value of *bx* from 0 to 7, assign RES[*by*][*bx*] the value 0.

(f) For each value of *by* from 0 to 7, and each value of *bx* from 0 to 7:

i. Assign P the value (PRED[*by*][*bx*] + RES[*by*][*bx*]).

ii. If P is greater than 255, assign P the value 255.

iii. If P is less than 0, assign P the value 0.

iv. If *pli* equals 0, assign **RECY**[BY + *by*][BX + *bx*] the value P.

v. Otherwise, if *pli* equals 1, assign **RECB**[BY + *by*][BX + *bx*] the value P.

vi. Otherwise, *pli* equals 2, so assign **RECR**[BY + *by*][BX + *bx*] the value P.

7.10 Loop Filtering

The loop filter is a simple deblocking filter that is based on running a small edge detecting filter over the coded block edges and adjusting the pixel values by a tapered response. The filter response is modulated by the following non-linear function:

$$\text{lflim}(\text{R}, \mathbf{L}) = \begin{cases} 0, & \text{R} \leq -2 * \mathbf{L} \\ -\text{R} - 2 * \mathbf{L}, & -2 * \mathbf{L} < \text{R} \leq -\mathbf{L} \\ \text{R}, & -\mathbf{L} < \text{R} < \mathbf{L} \\ -\text{R} + 2 * \mathbf{L}, & \mathbf{L} \leq \text{R} < 2 * \mathbf{L} \\ 0, & 2 * \mathbf{L} \leq \text{R} \end{cases}$$

Here **L** is a limiting value equal to **LFLIMS**[*qi0*]. It defines the peaks of the function. **LFLIMS** is an array of values specified in the setup header and is indexed by *qi0*, the first quantization index for the frame, the one used for all the DC coefficients. Larger values of **L** indicate a stronger filter.

7.10.1 Horizontal Filter

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RECP	2D Integer Array	8	No	A **RPH** $\times$ **RPW** array containing the contents of a plane of the reconstructed frame.
FX	Integer	20	No	The horizontal pixel index of the lower-left corner of the area to be filtered.
FY	Integer	20	No	The vertical pixel index of the lower-left corner of the area to be filtered.
L	Integer	7	No	The loop filter limit value.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RECP	2D Integer Array	8	No	A **RPH** $\times$ **RPW** array containing the contents of a plane of the reconstructed frame.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
R	Integer	9	Yes	The edge detector response.
P	Integer	9	Yes	A filtered pixel value.
by	Integer	20	No	The vertical pixel index in the block.

This procedure applies a 4-tap horizontal filter to each row of a vertical block edge.

1. For each value of *by* from 0 to 7:
   (a) Assign R the value

$$(\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}]-3*\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+1]+ \\ 3*\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+2]-\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+3]+4) >> 3$$

   (b) Assign P the value $(\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+1]+\mathrm{lflim}(\mathrm{R},\mathbf{L}))$.
   (c) If P is less than zero, assign $\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+1]$ the value zero.

(d) Otherwise, if P is greater than 255, assign $\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+1]$ the value 255.

(e) Otherwise, assign $\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+1]$ the value P.

(f) Assign P the value $(\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+2] - \text{lflim}(\text{R}, \mathbf{L}))$.

(g) If P is less than zero, assign $\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+2]$ the value zero.

(h) Otherwise, if P is greater than 255, assign $\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+2]$ the value 255.

(i) Otherwise, assign $\mathbf{RECP}[\mathbf{FY}+by][\mathbf{FX}+2]$ the value P.

7.10.2 Vertical Filter

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RECP	2D Integer Array	8	No	A **RPH** × **RPW** array containing the contents of a plane of the reconstructed frame.
FX	Integer	20	No	The horizontal pixel index of the lower-left corner of the area to be filtered.
FY	Integer	20	No	The vertical pixel index of the lower-left corner of the area to be filtered.
L	Integer	7	No	The loop filter limit value.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RECP	2D Integer Array	8	No	A **RPH** × **RPW** array containing the contents of a plane of the reconstructed frame.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
R	Integer	9	Yes	The edge detector response.

Name	Type	Size (bits)	Signed?	Description and restrictions
P	Integer	9	Yes	A filtered pixel value.
bx	Integer	20	No	The horizontal pixel index in the block.

This procedure applies a 4-tap vertical filter to each column of a horizontal block edge.

1. For each value of *bx* from 0 to 7:
   (a) Assign R the value

$$(\mathbf{RECP}[\mathbf{FY}][\mathbf{FX}+bx]-3*\mathbf{RECP}[\mathbf{FY}+1][\mathbf{FX}+bx]+ 3*\mathbf{RECP}[\mathbf{FY}+2][\mathbf{FX}+bx]-\mathbf{RECP}[\mathbf{FY}+3][\mathbf{FX}+bx]+4) >> 3$$

   (b) Assign P the value $(\mathbf{RECP}[\mathbf{FY}+1][\mathbf{FX}+bx]+\mathrm{lflim}(\mathrm{R},\mathbf{L}))$.
   (c) If P is less than zero, assign $\mathbf{RECP}[\mathbf{FY}+1][\mathbf{FX}+bx]$ the value zero.
   (d) Otherwise, if P is greater than 255, assign $\mathbf{RECP}[\mathbf{FY}+1][\mathbf{FX}+bx]$ the value 255.
   (e) Otherwise, assign $\mathbf{RECP}[\mathbf{FY}+1][\mathbf{FX}+bx]$ the value P.
   (f) Assign P the value $(\mathbf{RECP}[\mathbf{FY}+2][\mathbf{FX}+bx]-\mathrm{lflim}(\mathrm{R},\mathbf{L}))$.
   (g) If P is less than zero, assign $\mathbf{RECP}[\mathbf{FY}+2][\mathbf{FX}+bx]$ the value zero.
   (h) Otherwise, if P is greater than 255, assign $\mathbf{RECP}[\mathbf{FY}+2][\mathbf{FX}+bx]$ the value 255.
   (i) Otherwise, assign $\mathbf{RECP}[\mathbf{FY}+2][\mathbf{FX}+bx]$ the value P.

7.10.3 Complete Loop Filter

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
LFLIMS	Integer array	7	No	A 64-element array of loop filter limit values.
RPYW	Integer	20	No	The width of the Y' plane of the reconstruced frame in pixels.
RPYH	Integer	20	No	The height of the Y' plane of the reconstruced frame in pixels.

Name	Type	Size (bits)	Signed?	Description and restrictions
RPCW	Integer	20	No	The width of the C_b and C_r planes of the reconstruced frame in pixels.
RPCH	Integer	20	No	The height of the C_b and C_r planes of the reconstruced frame in pixels.
NBS	Integer	36	No	The total number of blocks in a frame.
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.
QIS	Integer array	6	No	An **NQIS**-element array of qi values.
RECY	2D Integer Array	8	No	A **RPYH** $\times$ **RPYW** array containing the contents of the Y' plane of the reconstructed frame.
RECCB	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_b plane of the reconstructed frame.
RECCR	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_r plane of the reconstructed frame.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RECY	2D Integer Array	8	No	A **RPYH** $\times$ **RPYW** array containing the contents of the Y' plane of the reconstructed frame.
RECCB	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_b plane of the reconstructed frame.
RECCR	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_r plane of the reconstructed frame.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
RPW	Integer	20	No	The width of the current plane of the reconstructed frame in pixels.
RPH	Integer	20	No	The height of the current plane of the reconstructed frame in pixels.
RECP	2D Integer Array	8	No	A **RPH** $\times$ **RPW** array containing the contents of the current plane of the reconstruced frame.
BX	Integer	20	No	The horizontal pixel index of the lower-left corner of the current block.
BY	Integer	20	No	The vertical pixel index of the lower-left corner of the current block.
FX	Integer	20	No	The horizontal pixel index of the lower-left corner of the area to be filtered.
FY	Integer	20	No	The vertical pixel index of the lower-left corner of the area to be filtered.
L	Integer	7	No	The loop filter limit value.
bi	Integer	36	No	The index of the current block in coded order.
bj	Integer	36	No	The index of a neighboring block in coded order.
pli	Integer	2	No	The color plane index of the current block.

This procedure defines the order that the various block edges are filtered. Because each application of one of the two filters above destructively modifies the contents of the reconstructed image, the precise output obtained differs depending on the order that horizontal and vertical filters are applied to the edges of a single block. The order defined here conforms to that used by VP3.

1. Assign L the value **LFLIMS**[**QIS**[0]].
2. For each block in *raster* order, with coded-order index *bi*:
   (a) If **BCODED**[*bi*] is non-zero:
      i. Assign *pli* the index of the color plane block *bi* belongs to.

pli	RECP	RPW	RPH
0	**RECY**	**RPYW**	**RPYH**
1	**RECCB**	**RPCW**	**RPCH**
2	**RECCR**	**RPCW**	**RPCH**

Table 7.85: Reconstructed Planes and Sizes for Each *pli*

ii. Assign RECP, RPW, and RPH the values given in Table 7.85 corresponding to the value of *pli*.

iii. Assign BX the horizontal pixel index of the lower-left corner of the block *bi*.

iv. Assign BY the vertical pixel index of the lower-left corner of the block *bi*.

v. If BX is greater than zero:

   A. Assign FX the value (BX − 2).

   B. Assign FY the value BY.

   C. Using RECP, FX, FY, and L, apply the horizontal block filter to the left edge of block *bi* with the procedure described in Section 7.10.1.

vi. If BY is greater than zero:

   A. Assign FX the value BX.

   B. Assign FY the value (BY − 2)

   C. Using RECP, FX, FY, and L, apply the vertical block filter to the bottom edge of block *bi* with the procedure described in Section 7.10.2.

vii. If (BX + 8) is less than RPW and **BCODED**[*bj*] is zero, where *bj* is the coded-order index of the block adjacent to *bi* on the right:

   A. Assign FX the value (BX + 6).

   B. Assign FY the value BY.

   C. Using RECP, FX, FY, and L, apply the horizontal block filter to the right edge of block *bi* with the procedure described in Section 7.10.1.

viii. If (BY + 8) is less than RPH and **BCODED**[*bj*] is zero, where *bj* is the coded-order index of the block adjacent to *bi* above:

   A. Assign FX the value BX.

   B. Assign FY the value (BY + 6)

   C. Using RECP, FX, FY, and L, apply the vertical block filter to the top edge of block *bi* with the procedure described in Section 7.10.2.

VP3 Compatibility The original VP3 decoder implemented unrestricted motion vectors by enlarging the reconstructed frame buffers and repeating the pixels on its edges into the padding region. However, for the previous reference frame this padding ocurred before the loop filter was applied, but for the golden reference frame it occurred afterwards.

This means that for the previous reference frame, the padding values were required to be stored separately from the main image values. Furthermore, even if the previous and golden reference frames were in fact the same frame, they could have different padding values. Finally, the encoder did not apply the loop filter at all, which resulted in artifacts, particularly in near-static scenes, due to prediction-loop mismatch. This last can only be considered a bug in the VP3 encoder.

Given all these things, Theora now uniformly applies the loop filter before the reference frames are padded. This means it is possible to use the same buffer for the previous and golden reference frames when they do indeed refer to the same frame. It also means that on architectures where memory bandwidth is limited, it is possible to avoid storing padding values, and simply clamp the motion vectors applied to each pixel as described in Sections 7.9.1 and 7.9.1. This means that the predicted pixel values along the edges of the frame might differ slightly between VP3 and Theora, but since the VP3 encoder did not apply the loop filter in the first place, this is not likely to impose any serious compatibility issues.

7.11 Complete Frame Decode

Input parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
FMBW	Integer	16	No	The width of the frame in macro blocks.
FMBH	Integer	16	No	The height of the frame in macro blocks.
NSBS	Integer	32	No	The total number of super blocks in a frame.
NBS	Integer	36	No	The total number of blocks in a frame.
NMBS	Integer	32	No	The total number of macro blocks in a frame.
FRN	Integer	32	No	The frame-rate numerator.
FRD	Integer	32	No	The frame-rate denominator.

Name	Type	Size (bits)	Signed?	Description and restrictions
PARN	Integer	24	No	The pixel aspect-ratio numerator.
PARD	Integer	24	No	The pixel aspect-ratio denominator.
CS	Integer	8	No	The color space.
PF	Integer	2	No	The pixel format.
NOMBR	Integer	24	No	The nominal bitrate of the stream, in bits per second.
QUAL	Integer	6	No	The quality hint.
KFGSHIFT	Integer	5	No	The amount to shift the key frame number by in the granule position.
LFLIMS	Integer array	7	No	A 64-element array of loop filter limit values.
ACSCALE	Integer array	16	No	A 64-element array of scale values for AC coefficients for each qi value.
DCSCALE	Integer array	16	No	A 64-element array of scale values for the DC coefficient for each qi value.
NBMS	Integer	10	No	The number of base matrices.
BMS	2D Integer array	8	No	A **NBMS** $\times 64$ array containing the base matrices.
NQRS	2D Integer array	6	No	A 2×3 array containing the number of quant ranges for a given qti and pli, respectively. This is at most 63.
QRSIZES	3D Integer array	6	No	A $2 \times 3 \times 63$ array of the sizes of each quant range for a given qti and pli, respectively. Only the first **NQRS**$[qti][pli]$ values will be used.

Name	Type	Size (bits)	Signed?	Description and restrictions
QRBMIS	3D Integer array	9	No	A $2 \times 3 \times 64$ array of the *bmi*'s used for each quant range for a given *qti* and *pli*, respectively. Only the first (**NQRS**$[qti][pli]+1$) values will be used.
HTS	Huffman table array			An 80-element array of Huffman tables with up to 32 entries each.
GOLDREFY	2D Integer Array	8	No	A **RPYH** $\times$ **RPYW** array containing the contents of the Y' plane of the golden reference frame.
GOLDREFCB	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_b plane of the golden reference frame.
GOLDREFCR	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_r plane of the golden reference frame.
PREVREFY	2D Integer Array	8	No	A **RPYH** $\times$ **RPYW** array containing the contents of the Y' plane of the previous reference frame.
PREVREFCB	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_b plane of the previous reference frame.
PREVREFCR	2D Integer Array	8	No	A **RPCH** $\times$ **RPCW** array containing the contents of the C_r plane of the previous reference frame.

Output parameters:

Name	Type	Size (bits)	Signed?	Description and restrictions
RECY	2D Integer Array	8	No	A **RPYH** × **RPYW** array containing the contents of the Y' plane of the reconstructed frame.
RECCB	2D Integer Array	8	No	A **RPCH** × **RPCW** array containing the contents of the C_b plane of the reconstructed frame.
RECCR	2D Integer Array	8	No	A **RPCH** × **RPCW** array containing the contents of the C_r plane of the reconstructed frame.
GOLDREFY	2D Integer Array	8	No	A **RPYH** × **RPYW** array containing the contents of the Y' plane of the golden reference frame.
GOLDREFCB	2D Integer Array	8	No	A **RPCH** × **RPCW** array containing the contents of the C_b plane of the golden reference frame.
GOLDREFCR	2D Integer Array	8	No	A **RPCH** × **RPCW** array containing the contents of the C_r plane of the golden reference frame.
PREVREFY	2D Integer Array	8	No	A **RPYH** × **RPYW** array containing the contents of the Y' plane of the previous reference frame.
PREVREFCB	2D Integer Array	8	No	A **RPCH** × **RPCW** array containing the contents of the C_b plane of the previous reference frame.

Name	Type	Size (bits)	Signed?	Description and restrictions
PREVREFCR	2D Integer Array	8	No	A **RPCH** × **RPCW** array containing the contents of the C_r plane of the previous reference frame.

Variables used:

Name	Type	Size (bits)	Signed?	Description and restrictions
FTYPE	Integer	1	No	The frame type.
NQIS	Integer	2	No	The number of *qi* values.
QIS	Integer array	6	No	An NQIS-element array of *qi* values.
BCODED	Integer Array	1	No	An **NBS**-element array of flags indicating which blocks are coded.
MBMODES	Integer Array	3	No	An **NMBS**-element array of coding modes for each macro block.
MVECTS	Array of 2D Integer Vectors	6	Yes	An **NBS**-element array of motion vectors for each block.
QIIS	Integer Array	2	No	An **NBS**-element array of *qii* values for each block.
COEFFS	2D Integer Array	16	Yes	An **NBS** × 64 array of quantized DCT coefficient values for each block in zig-zag order.
NCOEFFS	Integer Array	7	No	An **NBS**-element array of the coefficient count for each block.
RPYW	Integer	20	No	The width of the Y' plane of the reference frames in pixels.
RPYH	Integer	20	No	The height of the Y' plane of the reference frames in pixels.
RPCW	Integer	20	No	The width of the C_b and C_r planes of the reference frames in pixels.
RPCH	Integer	20	No	The height of the C_b and C_r planes of the reference frames in pixels.

Name	Type	Size (bits)	Signed?	Description and restrictions
bi	Integer	36	No	The index of the current block in coded order.

This procedure uses all the procedures defined in the previous section of this chapter to decode and reconstruct a complete frame. It takes as input values decoded from the headers, as well as the current reference frames. As output, it gives the uncropped, reconstructed frame. This should be cropped to picture region before display. As a special case, a 0-byte packet is treated exactly like an inter frame with no coded blocks.

1. If the size of the data packet is non-zero:
   (a) Decode the frame header values FTYPE, NQIS, and QIS using the procedure given in Section 7.1.
   (b) Using FTYPE, **NSBS**, and **NBS**, decode the list of coded block flags into BCODED using the procedure given in Section 7.3.
   (c) Using FTYPE, **NMBS**, **NBS**, and **BCODED**, decode the macro block coding modes into MBMODES using the procedure given in Section 7.4.
   (d) If FTYPE is non-zero (inter frame), using **PF**, **NMBS**, MBMODES, **NBS**, and BCODED, decode the motion vectors into MVECTS using the procedure given in Section 7.5.1.
   (e) Using **NBS**, BCODED, and NQIS, decode the block-level *qi* values into QIIS using the procedure given in Section 7.6.
   (f) Using **NBS**, **NMBS**, BCODED, and **HTS**, decode the DCT coefficients into NCOEFFS and NCOEFFS using the procedure given in Section 7.7.3.
   (g) Using BCODED and MBMODES, undo the DC prediction on the DC coefficients stored in COEFFS using the procedure given in Section 7.8.2.
2. Otherwise:
   (a) Assign FTYPE the value 1 (inter frame).
   (b) Assign NQIS the value 1.
   (c) Assign QIS[0] the value 63.
   (d) For each value of *bi* from 0 to (**NBS** − 1), assign BCODED[*bi*] the value zero.
3. Assign RPYW and RPYH the values (16 ∗ **FMBW**) and (16 ∗ **FMBH**), respectively.

PF	RPCW	RPCH
0	$8 * \mathbf{FMBW}$	$8 * \mathbf{FMBH}$
2	$8 * \mathbf{FMBW}$	$16 * \mathbf{FMBH}$
3	$16 * \mathbf{FMBW}$	$16 * \mathbf{FMBH}$

Table 7.89: Width and Height of Chroma Planes for each Pixel Format

4. Assign RPCW and RPCH the values from the row of Table 7.89 corresponding to **PF**.

5. Using **ACSCALE**, **DCSCALE**, **BMS**, **NQRS**, **QRSIZES**, **QRBMIS**, **NBS**, BCODED, MBMODES, MVECTS, COEFFS, NCOEFFS, QIS, QIIS, RPYW, RPYH, RPCW, RPCH, **GOLDREFY**, **GOLDREFCB**, **GOLDREFCR**, **PREVREFY**, **PREVREFCB**, and **PREVREFCR**, reconstruct the complete frame into **RECY**, **RECCB**, and **RECCR** using the procedure given in Section 7.9.4.

6. Using **LFLIMS**, RPYW, RPYH, RPCW, RPCH, **NBS**, BCODED, and QIS, apply the loop filter to the reconstructed frame in **RECY**, **RECCB**, and **RECCR** using the procedure given in Section 7.10.3.

7. If FTYPE is zero (intra frame), assign **GOLDREFY**, **GOLDREFCB**, and **GOLDREFCR** the values **RECY**, **RECCB**, and **RECCR**, respectively.

8. Assign **PREVREFY**, **PREVREFCB**, and **PREVREFCR** the values **RECY**, **RECCB**, and **RECCR**, respectively.

Appendix A

Ogg Bitstream Encapsulation

A.1 Overview

This document specifies the embedding or encapsulation of Theora packets in an Ogg transport stream.

Ogg is a stream oriented wrapper for coded, linear time-based data. It provides syncronization, multiplexing, framing, error detection and seeking landmarks for the decoder and complements the raw packet format used by the Theora codec.

This document assumes familiarity with the details of the Ogg standard. The Xiph.org documentation provides an overview of the Ogg transport stream format at `http://www.xiph.org/ogg/doc/oggstream.html` and a detailed description at `http://www.xiph.org/ogg/doc/framing.html`. The format is also defined in RFC 3533 [Pfe03]. While Theora packets can be embedded in a wide variety of media containers and streaming mechanisms, the Xiph.org Foundation recommends Ogg as the native format for Theora video in file-oriented storage and transmission contexts.

A.1.1 MIME type

The generic MIME type of any Ogg file is `application/ogg`. The specific MIME type for the Ogg Theora profile documented here is `video/ogg`. This is the MIME type recommended for files conforming to this appendix. The recommended filename extension is `.ogv`.

Outside of an encapsulation, the mime type `video/theora` may be used to refer specifically to the Theora compressed video stream.

A.2 Embedding in a logical bitstream

Ogg separates the concept of a *logical bitstream* consisting of the framing of a particular sequence of packets and complete within itself from the *physical bitstream* which may consist either of a single logical bitstream or a number of logical bitstreams multiplexed together. This section specifies the embedding of Theora packets in a logical Ogg bitstream. The mapping of Ogg Theora logical bitstreams into a multiplexed physical Ogg stream is described in the next section.

A.2.1 Headers

The initial identification header packet appears by itself in a single Ogg page. This page defines the start of the logical stream and MUST have the 'beginning of stream' flag set.

The second and third header packets (comment metadata and decoder setup data) can together span one or more Ogg pages. If there are additional non-normative header packets, they MUST be included in this sequence of pages as well. The comment header packet MUST begin the second Ogg page in the logical bitstream, and there MUST be a page break between the last header packet and the first frame data packet.

These two page break requirements facilitate stream identification and simplify header acquisition for seeking and live streaming applications.

All header pages MUST have their granule position field set to zero.

A.2.2 Frame data

The first frame data packet in a logical bitstream MUST begin a new Ogg page. All other data packets are placed one at a time into Ogg pages until the end of the stream. Packets can span pages and multiple packets can be placed within any one page. The last page in the logical bitstream SHOULD have its 'end of stream' flag set to indicate complete transmission of the available video.

Frame data pages MUST be marked with a granule position corresponding to the end of the display interval of the last frame/packet that finishes in that page. See the next section for details.

A.2.3 Granule position

Data packets are marked by a granulepos derived from the count of decodable frames after that packet is processed. The field itself is divided into two sections, the width of the less significant section being given by the KFGSHIFT parameter decoded from the identification header (Section 6.2). The more significant portion of the field gives the count of coded frames after the coding of the last keyframe in stream, and the less significant portion gives the count of frames since the last keyframe. Thus a stream would begin with a split granulepos of

1|0 (a keyframe), followed by 1|1, 1|2, 1|3, etc. Around a keyframe in the middle of the stream the granulepos sequence might be 1234|35, 1234|36, 1234|37, 1271|0 (for the keyframe), 1271|1, and so on. In this way the granulepos field increased monotonically as required by the Ogg format, but contains information necessary to efficiently find the previous keyframe to continue decoding after a seek.

Prior to bitstream version 3.2.1, data packets were marked by a granulepos derived from the index of the frame being decoded, rather than the count. That is they marked the beginning of the display interval of a frame rather than the end. Such streams have the VREV field of the identification header set to '0' instead of '1'. They can be interpreted according to the description above by adding 1 to the more signification field of the split granulepos when VREV is less than 1.

A.3 Multiplexed stream mapping

Applications supporting Ogg Theora must support Theora bitstreams multiplexed with compressed audio data in the Vorbis I and Speex formats, and should support Ogg-encapsulated MNG graphics for overlays.

Multiple audio and video bitstreams may be multiplexed together. How playback of multiple/alternate streams is handled is up to the application. Some conventions based on included metadata aide interoperability in this respect.

A.3.1 Chained streams

Ogg Theora decoders and playback applications MUST support both grouped streams (multiplexed concurrent logical streams) and chained streams (sequential concatenation of independent physical bitstreams).

The number and codec data types of multiplexed streams and the decoder parameters for those stream types that re-occur can all change at a chaining boundary. A playback application MUST be prepared to handle such changes and SHOULD do so smoothly with the minimum possible visible disruption. The specification of grouped streams below applies independently to each segment of a chained bitstream.

A.3.2 Grouped streams

At the beginning of a multiplexed stream, the 'beginning of stream' pages for each logical bitstream will be grouped together. Within these, the first page to occur MUST be the Theora page. This facilitates identification of Ogg Theora files among other Ogg-encapsulated content. A playback application must nevertheless handle streams where this arrangement is not correct.

If there is more than one Theora logical stream, the first page should be from the primary stream. That is, the best choice for the stream a generic player should begin displaying without special user direction. If there is more than

one audio stream, or of any other stream type, the identification page of the primary stream of that type should be placed before the others.

After the 'beginning of stream' pages, the header pages of each of the logical streams MUST be grouped together before any data pages occur.

After all the header pages have been placed, the data pages are multiplexed together. They should be placed in the stream in increasing order by the time equivalents of their granule position fields. This facilitates seeking while limiting the buffering requirements of the playback demultiplexer.

Appendix B

VP3

B.1 VP3 Compatibility

This section lists all of the encoder and decoder issues that may affect VP3 compatibly. Each is described in more detail in the text itself. This list is provided merely for reference.

- Bitstream headers (Section 6).
  - Identification header (Section 6.2).
    - Non-multiple of 16 picture sizes.
    - Standardized color spaces.
    - Support for $4:4:4$ and $4:2:2$ pixel formats.
  - Setup header
    - Loop filter limit values (Section 6.4.1).
    - Quantization parameters (Section 6.4.2).
    - Huffman tables (Section 6.4.4).
- Frame header format (Section 7.1).
- Extended long-run bit strings (Section 7.2.1).
- INTER_MV_FOUR handling of uncoded blocks (Section 7.5.2).
- Block-level qi values (Section 7.6).
- Zero-length EOB runs (Section 7.7.1).
- Unrestricted motion vector padding and the loop filter (Section 7.10.3).

B.2 Loop Filter Limit Values

The hard-coded loop filter limit values used in VP3 are defined as follows:

$$\begin{aligned}\textbf{LFLIMS} = \{&30, 25, 20, 20, 15, 15, 14, 14,\\ &13, 13, 12, 12, 11, 11, 10, 10,\\ &9, 9, 8, 8, 7, 7, 7, 7,\\ &6, 6, 6, 6, 5, 5, 5, 5,\\ &4, 4, 4, 4, 3, 3, 3, 3,\\ &2, 2, 2, 2, 2, 2, 2, 2,\\ &0, 0, 0, 0, 0, 0, 0, 0,\\ &0, 0, 0, 0, 0, 0, 0, 0\}\end{aligned}$$

B.3 Quantization Parameters

The hard-coded quantization parameters used by VP3 are defined as follows:

$$\begin{aligned}\textbf{ACSCALE} = \{&500, 450, 400, 370, 340, 310, 285, 265,\\ &245, 225, 210, 195, 185, 180, 170, 160,\\ &150, 145, 135, 130, 125, 115, 110, 107,\\ &100, 96, 93, 89, 85, 82, 75, 74,\\ &70, 68, 64, 60, 57, 56, 52, 50,\\ &49, 45, 44, 43, 40, 38, 37, 35,\\ &33, 32, 30, 29, 28, 25, 24, 22,\\ &21, 19, 18, 17, 15, 13, 12, 10\}\\ \textbf{DCSCALE} = \{&220, 200, 190, 180, 170, 170, 160, 160,\\ &150, 150, 140, 140, 130, 130, 120, 120,\\ &110, 110, 100, 100, 90, 90, 90, 80,\\ &80, 80, 70, 70, 70, 60, 60, 60,\\ &60, 50, 50, 50, 50, 40, 40, 40,\\ &40, 40, 30, 30, 30, 30, 30, 30,\\ &30, 20, 20, 20, 20, 20, 20, 20,\\ &20, 10, 10, 10, 10, 10, 10, 10\}\end{aligned}$$

VP3 defines only a single quantization range for each quantization type and color plane, and the base matrix used is constant throughout the range. There are three base matrices defined. The first is used for the Y' channel of INTRA mode blocks, and the second for both the C_b and C_r channels of INTRA mode blocks. The last is used for INTER mode blocks of all channels.

$$
\begin{aligned}
\mathbf{BMS} = \{\ &\{16,\ 11,\ 10,\ 16,\ 24,\ 40,\ 51,\ 61,\\
&\ 12,\ 12,\ 14,\ 19,\ 26,\ 58,\ 60,\ 55,\\
&\ 14,\ 13,\ 16,\ 24,\ 40,\ 57,\ 69,\ 56,\\
&\ 14,\ 17,\ 22,\ 29,\ 51,\ 87,\ 80,\ 62,\\
&\ 18,\ 22,\ 37,\ 58,\ 68,\ 109,\ 103,\ 77,\\
&\ 24,\ 35,\ 55,\ 64,\ 81,\ 104,\ 113,\ 92,\\
&\ 49,\ 64,\ 78,\ 87,\ 103,\ 121,\ 120,\ 101,\\
&\ 72,\ 92,\ 95,\ 98,\ 112,\ 100,\ 103,\ 99\},\\
&\{17,\ 18,\ 24,\ 47,\ 99,\ 99,\ 99,\ 99,\\
&\ 18,\ 21,\ 26,\ 66,\ 99,\ 99,\ 99,\ 99,\\
&\ 24,\ 26,\ 56,\ 99,\ 99,\ 99,\ 99,\ 99,\\
&\ 47,\ 66,\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\\
&\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\\
&\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\\
&\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\\
&\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\ 99,\ 99\},\\
&\{16,\ 16,\ 16,\ 20,\ 24,\ 28,\ 32,\ 40,\\
&\ 16,\ 16,\ 20,\ 24,\ 28,\ 32,\ 40,\ 48,\\
&\ 16,\ 20,\ 24,\ 28,\ 32,\ 40,\ 48,\ 64,\\
&\ 20,\ 24,\ 28,\ 32,\ 40,\ 48,\ 64,\ 64,\\
&\ 24,\ 28,\ 32,\ 40,\ 48,\ 64,\ 64,\ 64,\\
&\ 28,\ 32,\ 40,\ 48,\ 64,\ 64,\ 64,\ 96,\\
&\ 32,\ 40,\ 48,\ 64,\ 64,\ 64,\ 96,\ 128,\\
&\ 40,\ 48,\ 64,\ 64,\ 64,\ 96,\ 128,\ 128\}\ \}
\end{aligned}
$$

The remaining parameters simply assign these matrices to the proper quant ranges.

$$
\begin{aligned}
\mathbf{NQRS} &= \{\{1,1,1\},\{1,1,1\}\}\\
\mathbf{QRSIZES} &= \{\{\{1\},\{1\},\{1\}\},\{\{1\},\{1\},\{1\}\}\}\\
\mathbf{QRBMIS} &= \{\{\{0,0\},\{1,1\},\{1,1\}\},\{\{2,2\},\{2,2\},\{2,2\}\}\}
\end{aligned}
$$

B.4 Huffman Tables

The following tables contain the hard-coded Huffman codes used by VP3. There are 80 tables in all, each with a Huffman code for all 32 token values. The tokens are sorted by the most significant bits of their Huffman code. This is the same order in which they will be decoded from the setup header.

Huffman Code	Token Value
b000	20
b001	19
b01000	7
b010010	30
b0100110	1
b01001110	3
b010011110	28
b010011111000	26
b010011111001	8
b01001111101	6
b0100111111	29
b0101	14
b0110	12
b0111	17
b1000	13
b1001	21
b101000	24
b101001	23
b10101	16
b1011000	31
b10110010	25
b101100110	2
b1011001110	4
b10110011110	5
b10110011111	27
b101101	0
b10111	22
b1100	18
b11010	15
b11011	11
b1110	10
b1111	9

VP3.1 Huffman Table Number 0

Huffman Code	Token Value
b000	20
b001	19
b0100	11
b0101	12
b0110	14
b0111	17
b10000	0
b100010	7
b10001100	3
b1000110100000	29
b1000110100001	8
b100011010001	26
b10001101001	6
b10001101010	5
b10001101011	28
b100011011	25
b1000111	1
b1001	13
b1010	21
b10110	16
b10111	22
b1100	18
b1101	10
b1110	9
b11110	15
b1111100	30
b1111101	23
b1111110	24
b11111110	31
b1111111100	4
b1111111101	27
b111111111	2

VP3.1 Huffman Table Number 1

Huffman Code	Token Value
b000	20
b001	19
b0100	11
b0101	12
b0110	14
b0111	17
b100000	1
b1000010	30
b10000110	2
b10000111	3
b10001	16
b1001	21
b1010	13
b10110	0
b10111	22
b1100	18
b11010	15
b11011000	31
b1101100100	25
b11011001010	27
b11011001011	6
b11011001100	5
b110110011010	26
b11011001101100	29
b11011001101101	8
b1101100110111	28
b1101100111	4
b1101101	24
b1101110	7
b1101111	23
b1110	10
b1111	9

VP3.1 Huffman Table Number 2

Huffman Code	Token Value
b0000	0
b0001	17
b0010	14
b00110	21
b001110	7
b001111	23
b010	10
b011	9
b1000	11
b1001	12
b1010	20
b1011000	3
b101100100	25
b1011001010	6
b1011001011	5
b1011001100000	29
b1011001100001	8
b101100110001	28
b10110011001	26
b1011001101	27
b101100111	4
b101101	1
b10111	16
b1100	18
b1101	13
b1110	19
b1111000	22
b1111001	30
b1111010	24
b11110110	31
b11110111	2
b11111	15

VP3.1 Huffman Table Number 3

Huffman Code	Token Value
b0000	15
b000100000	5
b000100001	25
b00010001	22
b0001001	31
b000101	24
b000110	7
b000111000	27
b0001110010	6
b0001110011000	29
b0001110011001	8
b000111001101	28
b00011100111	26
b00011101	4
b0001111	2
b0010	17
b0011	0
b0100	14
b0101	11
b0110	12
b0111	19
b100	9
b101	10
b110000	21
b110001	23
b11001	16
b1101	18
b1110	13
b111100	1
b1111010	3
b1111011	30
b11111	20

VP3.1 Huffman Table Number 4

Huffman Code	Token Value
b0000	15
b00010	1
b000110	7
b000111	3
b0010	17
b0011	19
b0100	14
b0101	18
b01100	20
b011010000	27
b011010001	5
b0110100100000	29
b0110100100001	8
b011010010001	28
b01101001001	26
b0110100101	25
b0110100110	6
b0110100111	22
b0110101	21
b011011	23
b0111	12
b1000	11
b1001	0
b101	9
b110	10
b11100	16
b1110100	2
b1110101	30
b11101100	4
b11101101	31
b1110111	24
b1111	13

VP3.1 Huffman Table Number 5

Huffman Code	Token Value
b000	13
b0010	17
b0011	18
b010000	30
b010001	24
b010010	2
b010011000	27
b010011001	6
b01001101	21
b0100111	31
b0101	14
b01100	1
b011010	20
b011011	3
b01110	16
b01111	19
b1000	12
b1001	11
b1010	0
b101100	23
b1011010	7
b101101100	5
b1011011010	25
b1011011011000	8
b10110110110010	29
b10110110110011	22
b101101101101	28
b10110110111	26
b10110111	4
b10111	15
b110	10
b111	9

VP3.1 Huffman Table Number 6

Huffman Code	Token Value
b00	10
b01000	3
b01001	19
b010100	24
b0101010	7
b01010110	5
b010101110	21
b010101111	6
b01011	16
b0110	14
b011100	23
b011101	2
b01111	1
b1000	11
b1001	12
b1010000	20
b1010001	4
b1010010000	25
b101001000100	28
b1010010001010	8
b10100100010110	29
b10100100010111	22
b10100100011	26
b101001001	27
b10100101	31
b1010011	30
b10101	18
b10110	17
b10111	15
b1100	13
b1101	0
b111	9

VP3.1 Huffman Table Number 7

Huffman Code	Token Value
b00000	29
b00001	7
b0001	0
b0010	13
b001100	26
b001101	19
b00111	14
b0100	24
b0101	12
b0110	11
b011100	17
b011101	1
b01111	28
b100000	18
b100001	8
b10001	25
b1001000	20
b10010010	21
b10010011000	6
b10010011001	5
b1001001101	4
b100100111	22
b100101	15
b10011	31
b101	10
b110	9
b1110	23
b111100	27
b11110100	3
b11110101	2
b1111011	16
b11111	30

VP3.1 Huffman Table Number 8

Huffman Code	Token Value
b0000	30
b00010	7
b0001100000	6
b0001100001	5
b000110001	4
b00011001	22
b0001101	3
b000111	16
b0010	13
b0011	24
b010000	19
b010001	26
b01001	14
b0101	0
b0110	12
b0111	11
b1000000	2
b1000001	20
b100001	17
b10001	25
b100100	18
b100101	15
b10011	31
b101	10
b110	9
b1110	23
b111100	1
b11110100	21
b11110101	8
b1111011	29
b111110	28
b111111	27

VP3.1 Huffman Table Number 9

Huffman Code	Token Value
b0000000	22
b0000001	8
b000001	2
b00001	31
b0001	24
b001000	29
b001001	3
b00101	25
b00110	30
b00111	1
b0100	23
b010100	16
b010101	7
b010110	19
b010111	26
b0110	13
b0111	12
b1000	11
b10010	14
b1001100000	6
b1001100001	5
b100110001	4
b10011001	21
b1001101	20
b100111	17
b1010	0
b101100	28
b101101	18
b101110	27
b101111	15
b110	10
b111	9

VP3.1 Huffman Table Number 10

Huffman Code	Token Value
b000	0
b0010000	4
b0010001	21
b001001	30
b00101	15
b00110	25
b001110	29
b0011110	7
b0011111000	6
b0011111001	5
b001111101	22
b00111111	8
b0100	23
b010100	26
b010101	19
b010110	16
b010111	2
b0110	13
b01110	1
b01111	14
b1000	12
b1001	11
b1010000	20
b1010001	31
b101001	17
b101010	3
b101011	18
b101100	27
b101101	28
b10111	24
b110	10
b111	9

VP3.1 Huffman Table Number 11

Huffman Code	Token Value
b00	9
b010	0
b01100	14
b01101	3
b011100	26
b011101	18
b011110	17
b01111100	8
b01111101	21
b0111111	30
b1000	12
b1001	11
b101000	15
b10100100	7
b1010010100	6
b1010010101	5
b101001011	22
b1010011	4
b101010	28
b101011	27
b10110	24
b101110	25
b101111	2
b11000	1
b11001	23
b1101000	29
b1101001	19
b1101010	16
b11010110	31
b11010111	20
b11011	13
b111	10

VP3.1 Huffman Table Number 12

Huffman Code	Token Value
b00	9
b010	0
b01100	2
b01101	14
b01110	24
b011110	17
b0111110	29
b01111110	21
b01111111	5
b1000	12
b1001	11
b101000	28
b101001	4
b101010	15
b101011	27
b10110	23
b101110	25
b1011110000	6
b1011110001	22
b101111001	8
b10111101	30
b1011111	19
b11000	3
b1100100	16
b1100101	26
b110011000	7
b110011001	31
b11001101	20
b1100111	18
b11010	13
b11011	1
b111	10

VP3.1 Huffman Table Number 13

Huffman Code	Token Value
b00	9
b010	0
b0110	3
b0111	1
b1000	12
b1001	11
b10100	23
b101010	15
b10101100	30
b10101101	21
b101011100	7
b101011101	6
b101011110	31
b1010111110	22
b1010111111	8
b10110	2
b1011100	5
b1011101	19
b1011110	16
b1011111	26
b11000	13
b1100100	18
b11001010	29
b11001011	20
b110011	24
b110100	14
b1101010	17
b1101011	28
b110110	4
b1101110	25
b1101111	27
b111	10

VP3.1 Huffman Table Number 14

Huffman Code	Token Value
b00	10
b01	9
b1000	12
b1001	11
b101000	15
b101001	5
b101010000	30
b101010001	29
b10101001	28
b101010100000	22
b101010100001	8
b10101010001	7
b1010101001	31
b101010101	21
b10101011	26
b1010110	19
b1010111	16
b1011	3
b11000	2
b11001	4
b1101000	18
b1101001	24
b1101010	17
b11010110	6
b11010111	25
b11011	13
b111000	14
b11100100	27
b11100101	20
b1110011	23
b11101	1
b1111	0

VP3.1 Huffman Table Number 15

Huffman Code	Token Value
b0000	15
b0001	11
b0010	12
b0011	21
b01000	0
b0100100	26
b0100101	1
b010011	24
b01010	22
b01011	30
b0110	14
b0111	10
b1000	9
b1001	17
b1010	13
b10110	23
b1011100	28
b1011101	25
b10111100	27
b101111010	2
b10111101100	29
b1011110110100	5
b10111101101010	8
b10111101101011	6
b101111011011	4
b1011110111	3
b1011111	31
b1100	20
b1101	18
b11100	16
b11101	7
b1111	19

VP3.1 Huffman Table Number 16

Huffman Code	Token Value
b0000	15
b0001	7
b0010	11
b0011	12
b010000	1
b010001	31
b0100100	26
b01001010	27
b01001011	2
b010011	22
b0101	17
b0110	14
b01110	30
b01111	0
b1000	9
b1001	10
b1010	20
b1011	13
b110000	24
b1100010	25
b11000110	3
b110001110000	6
b110001110001	5
b110001110010	29
b110001110011	8
b1100011101	4
b110001111	28
b11001	21
b1101	18
b11100	16
b11101	23
b1111	19

VP3.1 Huffman Table Number 17

Huffman Code	Token Value
b00000	21
b000010	25
b000011	1
b0001	15
b0010	20
b0011	7
b0100	11
b0101	12
b0110	17
b0111	14
b1000000	3
b1000001	22
b100001	31
b100010	24
b10001100	27
b10001101	2
b100011100000	6
b1000111000010	29
b1000111000011	8
b10001110001	5
b1000111001	4
b100011101	28
b10001111	26
b1001	10
b1010	9
b1011	19
b1100	18
b11010	30
b11011	0
b1110	13
b11110	16
b11111	23

VP3.1 Huffman Table Number 18

Huffman Code	Token Value
b0000000	28
b0000001	27
b00000100	22
b000001010000	8
b000001010001	6
b00000101001	29
b0000010101	5
b000001011	4
b0000011	2
b000010	21
b000011	1
b0001	15
b0010	23
b0011	7
b0100	11
b0101	17
b0110	12
b0111	19
b100000	25
b1000010	26
b1000011	3
b10001	20
b1001	18
b1010	14
b101100	31
b101101	24
b10111	30
b1100	10
b1101	9
b1110	13
b11110	16
b11111	0

VP3.1 Huffman Table Number 19

Huffman Code	Token Value
b0000	30
b0001	15
b0010	17
b0011	0
b0100	7
b0101	18
b0110	23
b0111000	21
b0111001	27
b0111010	2
b0111011	26
b011110	25
b011111	1
b1000	12
b1001	11
b1010	14
b10110	16
b10111000	28
b1011100100	5
b10111001010	22
b1011100101100	8
b1011100101101	6
b101110010111	29
b101110011	4
b1011101	3
b101111	20
b1100	13
b11010	19
b110110	31
b110111	24
b1110	10
b1111	9

VP3.1 Huffman Table Number 20

Huffman Code	Token Value
b000	9
b0010	30
b001100	3
b0011010	28
b0011011	27
b00111	31
b0100	7
b01010	24
b01011	19
b0110	0
b0111	12
b1000	11
b1001	14
b1010	23
b10110	16
b101110000	21
b10111000100	6
b1011100010100	22
b1011100010101	8
b101110001011	29
b1011100011	5
b10111001	4
b1011101	2
b1011110	20
b1011111	26
b1100	13
b11010	18
b110110	25
b110111	1
b11100	17
b11101	15
b1111	10

VP3.1 Huffman Table Number 21

Huffman Code	Token Value
b000	10
b001	9
b01000	18
b01001	25
b010100	26
b010101	19
b01011	1
b01100	31
b01101	17
b0111	14
b10000	24
b100010	3
b1000110000	6
b100011000100	8
b1000110001010	22
b1000110001011	21
b10001100011	29
b100011001	5
b10001101	20
b1000111	27
b1001	12
b1010	11
b1011	13
b1100	0
b1101	23
b11100	15
b11101	7
b11110000	4
b11110001	28
b1111001	2
b111101	16
b11111	30

VP3.1 Huffman Table Number 22

Huffman Code	Token Value
b000	0
b001	10
b010	9
b01100	3
b011010	27
b011011	16
b0111	13
b10000	31
b100010	17
b1000110	4
b1000111	28
b1001	11
b1010	12
b10110	24
b10111	7
b11000	25
b110010	26
b110011	2
b11010	1
b11011	14
b1110	23
b11110000	19
b1111000100000	20
b1111000100001	8
b1111000100010	22
b1111000100011	21
b11110001001	29
b1111000101	6
b111100011	5
b1111001	18
b111101	15
b11111	30

VP3.1 Huffman Table Number 23

Huffman Code	Token Value
b000	9
b0010	24
b0011	7
b01000	17
b010010	19
b0100110	20
b01001110	2
b010011110	3
b01001111100	4
b0100111110100	6
b0100111110101	5
b010011111011	22
b0100111111	21
b0101	14
b01100	25
b01101	15
b011100	27
b011101	29
b01111	28
b1000	30
b1001	13
b1010	12
b1011	11
b1100000	8
b1100001	1
b110001	16
b11001	31
b1101	23
b111000	18
b111001	26
b11101	0
b1111	10

VP3.1 Huffman Table Number 24

Huffman Code	Token Value
b000	10
b001	9
b010000	27
b0100010	20
b010001100000	6
b010001100001	5
b01000110001	22
b0100011001	4
b010001101	21
b01000111	8
b01001	25
b0101	14
b011000	19
b011001	1
b01101	15
b0111	0
b1000	30
b1001	13
b10100	31
b1010100	29
b10101010	3
b10101011	2
b101011	26
b1011	12
b1100	11
b110100	28
b110101	16
b11011	7
b1110	23
b111100	18
b111101	17
b11111	24

VP3.1 Huffman Table Number 25

Huffman Code	Token Value
b000	9
b001000	2
b0010010	8
b001001100000	22
b001001100001	6
b00100110001	5
b0010011001	21
b001001101	4
b00100111	20
b00101	1
b00110	15
b00111	26
b0100	24
b010100	29
b010101	18
b01011	28
b0110	13
b011100	16
b011101	27
b01111	25
b1000	30
b1001	12
b1010	11
b101100	17
b1011010	19
b1011011	3
b10111	31
b1100	0
b11010	7
b11011	14
b1110	23
b1111	10

VP3.1 Huffman Table Number 26

Huffman Code	Token Value
b0000	12
b0001	11
b001	10
b010	9
b011	23
b10000	7
b10001	14
b100100	3
b10010100000	6
b100101000010	22
b100101000011	21
b1001010001	5
b100101001	20
b10010101	4
b1001011	18
b10011	1
b1010	24
b101100	15
b101101	29
b10111	28
b11000	26
b11001000	8
b11001001	19
b1100101	16
b110011	27
b11010	13
b11011	30
b11100	25
b1110100	17
b1110101	2
b111011	31
b1111	0

VP3.1 Huffman Table Number 27

Huffman Code	Token Value
b000	10
b001	9
b0100	25
b0101000	4
b0101001	18
b0101010	16
b0101011	17
b01011	28
b011	0
b100	23
b1010	24
b101100	29
b101101	2
b10111	13
b11000	26
b11001	30
b11010	1
b110110	27
b110111	7
b111000	3
b11100100	8
b1110010100000	22
b1110010100001	21
b111001010001	6
b11100101001	20
b1110010101	5
b111001011	19
b1110011	15
b111010	14
b111011	31
b11110	12
b11111	11

VP3.1 Huffman Table Number 28

Huffman Code	Token Value
b000	10
b001	9
b0100	1
b01010	13
b010110	29
b010111	7
b011	23
b100	0
b1010	24
b10110	30
b10111	3
b11000	28
b110010	14
b110011	31
b11010	12
b11011	11
b11100	26
b1110100	15
b1110101	4
b111011	27
b11110	25
b11111000	16
b11111001	17
b111110100000	20
b1111101000010	22
b1111101000011	21
b11111010001	6
b1111101001	19
b111110101	5
b111110110	8
b111110111	18
b111111	2

VP3.1 Huffman Table Number 29

Huffman Code	Token Value
b000	10
b001	9
b010	23
b0110000000	19
b01100000010	20
b011000000110	22
b011000000111	21
b011000001	18
b01100001	17
b0110001	5
b011001	14
b01101	30
b0111	1
b100	0
b1010	24
b10110	28
b1011100	15
b10111010	16
b101110110	8
b101110111	6
b101111	31
b11000	2
b11001	12
b11010	11
b110110	4
b110111	27
b11100	26
b111010	13
b1110110	29
b1110111	7
b11110	3
b11111	25

VP3.1 Huffman Table Number 30

Huffman Code	Token Value
b000	0
b001	10
b010	9
b0110	24
b0111000	29
b0111001000	17
b0111001001000	22
b0111001001001	21
b0111001001010	18
b01110010010110	20
b01110010010111	19
b01110010011	8
b011100101	16
b01110011	15
b011101	27
b01111	12
b100	23
b1010	1
b10110	11
b101110	13
b1011110	7
b1011111	14
b1100	3
b11010	2
b11011	26
b111000	28
b111001	5
b11101	4
b1111000	6
b1111001	31
b111101	30
b11111	25

VP3.1 Huffman Table Number 31

Huffman Code	Token Value
b00000	24
b000010	28
b000011	21
b0001	23
b0010	7
b0011	15
b0100	17
b010100	25
b01010100	2
b010101010	22
b010101011	8
b0101011	1
b01011	0
b0110	19
b0111	11
b1000	12
b1001	9
b1010	10
b1011	18
b1100	14
b11010	20
b1101100	26
b11011010	27
b110110110000	6
b110110110001	5
b11011011001	4
b1101101101	29
b110110111	3
b110111	31
b11100	30
b11101	16
b1111	13

VP3.1 Huffman Table Number 32

Huffman Code	Token Value
b0000	30
b000100	1
b000101	28
b00011	24
b0010	17
b0011	15
b0100	18
b0101	23
b01100	31
b0110100	27
b01101010	3
b01101011	21
b011011	25
b0111	7
b1000	12
b1001	11
b1010	14
b101100	20
b1011010	26
b10110110	2
b1011011100000	6
b1011011100001	5
b101101110001	22
b10110111001	4
b1011011101	29
b101101111	8
b10111	16
b1100	9
b1101	10
b11100	19
b11101	0
b1111	13

VP3.1 Huffman Table Number 33

Huffman Code	Token Value
b000	13
b0010	15
b0011	0
b0100	30
b01010	24
b01011	31
b0110	23
b0111	7
b1000000	20
b10000010	8
b1000001100	4
b100000110100	5
b1000001101010	22
b1000001101011	6
b10000011011	21
b100000111	29
b100001	28
b10001	16
b1001	14
b10100000	3
b10100001	2
b1010001	27
b101001	25
b10101	18
b1011	11
b1100	12
b1101	10
b1110	9
b11110	17
b111110	19
b1111110	26
b1111111	1

VP3.1 Huffman Table Number 34

Huffman Code	Token Value
b0000	30
b00010	18
b00011	16
b001	9
b010	10
b01100	31
b011010	26
b011011	1
b0111	0
b1000	14
b10010	17
b10011	24
b1010	23
b1011	11
b1100	12
b1101	13
b11100	15
b11101000000	5
b1110010000010	6
b1110100000110	22
b1110100000111	21
b1110100001	4
b111010001	20
b11101001	3
b1110101	19
b111011	25
b111100	28
b1111010	27
b11110110	2
b1111101110	29
b1111101111	8
b11111	7

VP3.1 Huffman Table Number 35

Huffman Code	Token Value
b0000	30
b0001	14
b001	9
b010	10
b01100	25
b011010	18
b0110110000	20
b01101100010	5
b011011000110	6
b0110110001110	22
b0110110001111	21
b011011001	4
b01101101	29
b0110111	3
b01110	31
b01111	15
b100000	27
b1000010	2
b10000110	8
b10000111	19
b10001	28
b100100	26
b100101	16
b10011	24
b1010	13
b1011	12
b1100	11
b1101	0
b1110	23
b111100	17
b111101	1
b11111	7

VP3.1 Huffman Table Number 36

Huffman Code	Token Value
b000	0
b0010	30
b00110	31
b00111	25
b010	9
b011	10
b1000	13
b10010	1
b10011	7
b101000	27
b10100100	29
b10100101	8
b1010011000	19
b1010011001000	20
b1010011001001	6
b1010011001010	22
b1010011001011	21
b10100110011	5
b101001101	4
b10100111	18
b101010	26
b101011	15
b1011	11
b1100	12
b11010	14
b11011	28
b11100	24
b1110100	17
b1110101	16
b1110110	2
b1110111	3
b1111	23

VP3.1 Huffman Table Number 37

Huffman Code	Token Value
b000	23
b00100	7
b00101	31
b00110	14
b00111	25
b010	0
b011	10
b100	9
b101000000	18
b101000001000	22
b101000001001	21
b101000001010	6
b1010000010110	20
b1010000010111	19
b1010000011	5
b10100001	8
b10100010	17
b10100011	16
b101001	27
b101010	26
b101011	2
b1011	11
b1100	12
b11010	1
b11011	30
b11100	28
b111010	3
b11101100	29
b11101101	4
b1110111	15
b11110	24
b11111	13

VP3.1 Huffman Table Number 38

Huffman Code	Token Value
b000	23
b0010	1
b00110	13
b00111000	15
b001110010	8
b001110011000	18
b0011100110010	20
b0011100110011	19
b0011100110100	22
b0011100110101	21
b001110011011	17
b00111001110	16
b00111001111	6
b0011101	7
b001111	27
b010	0
b0110	11
b0111	12
b100	9
b101	10
b11000	2
b11001	30
b110100	26
b110101	4
b11011	25
b111000	31
b11100100	5
b11100101	29
b1110011	14
b11101	3
b11110	28
b11111	24

VP3.1 Huffman Table Number 39

Huffman Code	Token Value
b000	10
b001	9
b01000	26
b01001	15
b0101	24
b0110	7
b011100	16
b011101	17
b01111	25
b1000	30
b1001	13
b1010000	1
b1010001	8
b101001	27
b10101	31
b10110	0
b10111000	19
b101110010	2
b1011100110000	22
b1011100110001	21
b1011100110010	4
b10111001100110	6
b10111001100111	5
b10111001101	20
b1011100111	3
b1011101	18
b101111	29
b1100	12
b1101	11
b11100	14
b11101	28
b1111	23

VP3.1 Huffman Table Number 40

Huffman Code	Token Value
b000	9
b001	23
b0100	28
b0101	24
b0110	13
b0111	30
b1000000	2
b1000001	18
b100001	1
b10001	14
b1001	0
b10100	25
b101010	15
b1010110000	4
b1010110001000	6
b1010110001001	5
b1010110001010	22
b1010110001011	21
b10101100011	20
b101011001	19
b10101101	3
b1010111	16
b10110	31
b101110	27
b1011110	17
b1011111	8
b1100	12
b1101	11
b11100	7
b111010	29
b111011	26
b1111	10

VP3.1 Huffman Table Number 41

Huffman Code	Token Value
b000	9
b0010	30
b0011000	17
b001100100	4
b001100101000	22
b001100101001	21
b001100101010	5
b0011001010110	20
b0011001010111	6
b0011001011	19
b00110011	18
b001101	8
b00111	1
b010	23
b0110	24
b01110	26
b01111	29
b10000	31
b1000100	16
b1000101	3
b1000110	2
b1000111	15
b1001	28
b1010	11
b1011	12
b11000	7
b11001	25
b11010	13
b110110	14
b110111	27
b1110	0
b1111	10

VP3.1 Huffman Table Number 42

Huffman Code	Token Value
b000	23
b001	10
b010	9
b011	0
b10000	27
b100010	14
b100011	2
b1001	24
b10100	13
b10101	26
b10110	30
b10111	29
b11000	1
b1100100	15
b110010100	4
b11001010100	19
b1100101010100	22
b1100101010101	21
b1100101010110	5
b11001010101110	20
b11001010101111	6
b1100101011	18
b110010110	17
b110010111	16
b110011	31
b1101	28
b11100	25
b111010	7
b1110110	8
b1110111	3
b11110	12
b11111	11

VP3.1 Huffman Table Number 43

Huffman Code	Token Value
b000	23
b001	10
b010	9
b0110	1
b0111	24
b10000	3
b10001	26
b1001000	4
b10010010	15
b100100110	16
b1001001110	17
b10010011110	18
b10010011111000	22
b10010011111001	21
b10010011111010	6
b100100111110110	20
b100100111110111	19
b100100111111	5
b100101	31
b10011	29
b101	0
b11000	25
b110010	7
b1100110	14
b1100111	8
b110100	13
b110101	30
b11011	11
b1110	28
b11110	12
b111110	2
b111111	27

VP3.1 Huffman Table Number 44

Huffman Code	Token Value
b000	28
b001	9
b010	10
b0110	24
b011100	4
b01110100	15
b011101010	5
b0111010110	16
b0111010111000	22
b0111010111001	21
b01110101110100	18
b01110101110101	6
b01110101110110	20
b01110101110111	19
b01110101111	17
b0111011	14
b011110	7
b011111	13
b1000	1
b10010	2
b10011	25
b101	0
b11000	29
b110010	30
b1100110	8
b1100111	31
b11010	12
b11011	11
b11100	3
b111010	27
b111011	26
b1111	23

VP3.1 Huffman Table Number 45

Huffman Code	Token Value
b000	28
b001	10
b010	9
b011000	13
b011001	30
b01101	4
b01110	25
b01111	29
b100	0
b1010	1
b10110	12
b10111	11
b1100	3
b110100000	15
b11010000100	6
b110100001010	18
b1101000010110	20
b1101000010111	19
b11010000110	16
b1101000011100	22
b1101000011101	21
b110100001111	17
b11010001	14
b1101001	31
b110101	26
b11011	2
b111000	27
b1110010	7
b11100110	5
b11100111	8
b11101	24
b1111	23

VP3.1 Huffman Table Number 46

Huffman Code	Token Value
b000	3
b00100	25
b001010000	14
b001010001	6
b0010100100	15
b001010010100	16
b0010100101010	18
b0010100101011	17
b0010100101100	20
b0010100101101	19
b0010100101110	22
b0010100101111	21
b001010011	8
b0010101	13
b001011	29
b0011	4
b010	10
b011	0
b100	9
b101000	26
b101001	27
b10101	12
b10110	11
b101110	5
b10111100	7
b10111101	31
b1011111	30
b1100	1
b11010	24
b11011	2
b1110	23
b1111	28

VP3.1 Huffman Table Number 47

Huffman Code	Token Value
b000	9
b001000	29
b001001	1
b0010100	20
b0010101	8
b001011	26
b0011	0
b0100	7
b01010	16
b01011	24
b01100	31
b01101	18
b0111	30
b1000	23
b1001	14
b10100	17
b101010	19
b101011	25
b1011	11
b1100	12
b11010000	2
b1101000100	4
b110100010100	5
b1101000101010	22
b1101000101011	6
b11010001011	21
b110100011	3
b1101001	27
b110101	28
b11011	15
b1110	13
b1111	10

VP3.1 Huffman Table Number 48

Huffman Code	Token Value
b000	10
b001	9
b0100	14
b010100	18
b010101	27
b01011	28
b0110000	3
b011000100000	6
b011000100001	5
b011000100010	22
b011000100011	21
b0110001001	20
b011000101	4
b01100011	19
b0110010	2
b0110011	8
b01101	15
b0111	30
b10000	31
b100010	26
b100011	29
b10010	24
b100110	16
b100111	17
b1010	0
b1011	23
b1100	13
b1101	12
b1110	11
b111100	1
b111101	25
b11111	7

VP3.1 Huffman Table Number 49

Huffman Code	Token Value
b000	10
b001	9
b010000	3
b01000100	4
b0100010100	5
b010001010100	20
b010001010101	6
b010001010110	22
b010001010111	21
b010001011	19
b0100011	8
b01001	15
b01010	25
b010110	17
b010111	16
b01100	1
b01101	28
b0111	30
b100000	27
b1000010	18
b1000011	2
b10001	31
b1001	13
b101000	29
b101001	26
b10101	24
b1011	23
b11000	7
b11001	14
b1101	12
b1110	11
b1111	0

VP3.1 Huffman Table Number 50

Huffman Code	Token Value
b0000000	8
b0000001	17
b000001	15
b00001	31
b00010	7
b00011	25
b001	0
b010	9
b011	10
b1000	13
b10010	14
b10011	28
b10100000	4
b10100001000	19
b1010000100100	20
b1010000100101	6
b1010000100110	22
b1010000100111	21
b1010000101	5
b101000011	18
b1010001	16
b101001	2
b101010	3
b101011	27
b1011	12
b1100	11
b11010	1
b110110	29
b110111	26
b11100	24
b11101	30
b1111	23

VP3.1 Huffman Table Number 51

Huffman Code	Token Value
b000	23
b00100	26
b00101000	17
b0010100100	18
b00101001010	6
b0010100101100	20
b0010100101101	19
b0010100101110	22
b0010100101111	21
b001010011	5
b00101010	8
b00101011	16
b0010110	4
b0010111	15
b00110	3
b00111	25
b010	9
b011	10
b100	0
b10100	28
b10101	30
b1011	11
b1100	12
b110100	7
b110101	27
b110110	29
b110111	14
b11100	13
b11101	24
b111100	31
b111101	2
b11111	1

VP3.1 Huffman Table Number 52

Huffman Code	Token Value
b000	23
b001000	7
b001001	4
b00101	30
b00110	25
b00111	2
b010	10
b011	9
b100	0
b1010	1
b101100	29
b101101	31
b10111	13
b1100	12
b1101	11
b111000	27
b111001	26
b11101	3
b11110	24
b111110000	8
b1111100010000	22
b1111100010001	21
b1111100010010	18
b11111000100110	20
b11111000100111	19
b11111000101	17
b11111000110	16
b11111000111	6
b111110010	15
b111110011	5
b1111101	14
b111111	28

VP3.1 Huffman Table Number 53

Huffman Code	Token Value
b000	23
b001000	31
b0010010	7
b00100110	14
b00100111000000	20
b00100111000001	19
b00100111000010	22
b00100111000011	21
b00100111000100	16
b00100111000101	8
b00100111000110	18
b00100111000111	17
b0010011101	15
b001001111	6
b00101	4
b0011	3
b010	0
b01100	25
b0110100	29
b0110101	5
b011011	30
b0111	1
b100	10
b101	9
b11000	2
b110010	28
b110011	13
b1101	11
b1110	12
b111100	27
b111101	26
b11111	24

VP3.1 Huffman Table Number 54

Huffman Code	Token Value
b000	0
b0010	4
b00110	24
b00111	5
b0100	1
b01010	25
b0101100	26
b0101101	31
b010111	27
b011	23
b100	10
b101	9
b1100	12
b1101	11
b11100	2
b11101000	7
b1110100100	30
b1110100101000	22
b1110100101001	21
b1110100101010	8
b11101001010110	16
b11101001010111	15
b1110100101100	14
b11101001011100	18
b11101001011101	17
b11101001011110	20
b11101001011111	19
b1110100111	29
b1110101	6
b1110110	28
b1110111	13
b1111	3

VP3.1 Huffman Table Number 55

Huffman Code	Token Value
b00000	26
b00001	29
b0001	24
b001	9
b010	10
b0110	30
b0111	13
b100000	8
b100001000000	22
b100001000001	21
b100001000010	5
b1000010000110	20
b1000010000111	6
b1000010001	4
b100001001	19
b10000101	3
b1000011	17
b10001	25
b100100	1
b100101	15
b10011	14
b1010	0
b10110	31
b101110	27
b1011110	16
b10111110	18
b10111111	2
b11000	7
b11001	28
b1101	12
b1110	11
b1111	23

VP3.1 Huffman Table Number 56

Huffman Code	Token Value
b000	9
b001	10
b010	0
b0110	24
b01110	26
b01111	1
b1000	28
b10010	7
b10011	25
b1010	11
b1011	12
b1100000	15
b1100001	3
b110001	14
b11001	30
b11010	13
b110110	8
b11011100	16
b1101110100	4
b1101110101000	5
b11011101010010	20
b11011101010011	6
b11011101010100	22
b11011101010101	21
b1101110101011	19
b11011101011	18
b110111011	17
b1101111	2
b1110	23
b11110	29
b111110	27
b111111	31

VP3.1 Huffman Table Number 57

Huffman Code	Token Value
b000	9
b001	10
b01000	27
b01001	30
b01010	26
b01011	13
b011	0
b1000	29
b100100	3
b100101	2
b10011	25
b1010	12
b1011	11
b1100	28
b1101	23
b11100	1
b111010	31
b11101100	15
b111011010	4
b1110110110	16
b11101101110	17
b111011011111000	22
b111011011111001	21
b1110110111101	5
b111011011111100	6
b1110110111111010	20
b1110110111111011	19
b1110110111111	18
b1110111	14
b111100	7
b111101	8
b11111	24

VP3.1 Huffman Table Number 58

Huffman Code	Token Value
b0000	12
b0001	11
b00100	2
b00101	26
b0011	1
b010	9
b011	10
b10000	3
b100010	30
b1000110	14
b100011100	15
b1000111010000	18
b1000111010001	6
b1000111010010	20
b1000111010011	19
b10001110101	5
b1000111011000	22
b1000111011001	21
b100011101101	17
b10001110111	16
b10001111	4
b10010	25
b100110	13
b100111	8
b101	0
b1100	28
b1101	23
b1110	29
b1111000	31
b1111001	7
b111101	27
b11111	24

VP3.1 Huffman Table Number 59

Huffman Code	Token Value
b0000	3
b00010	25
b000110	4
b0001110	30
b0001111	7
b001	29
b010	10
b011	9
b1000	23
b1001	28
b101	0
b1100	1
b110100	8
b110101	27
b11011	2
b11100	24
b11101	12
b11110	11
b111110000	14
b1111100010	5
b11111000110	15
b111110001110000	20
b111110001110001	19
b111110001110010	22
b111110001110011	21
b111110001110100	16
b111110001110101	6
b111110001110110	18
b111110001110111	17
b11111001	31
b1111101	13
b111111	26

VP3.1 Huffman Table Number 60

Huffman Code	Token Value
b0000	23
b0001	2
b001	29
b0100000	13
b01000010	31
b01000011	30
b010001	27
b01001	24
b0101	28
b01100	12
b01101	11
b011100000	5
b0111000010	14
b0111000011000	18
b0111000011001	17
b0111000011010	20
b0111000011011	19
b0111000011100	22
b0111000011101	21
b0111000011110	6
b01110000111110	16
b01110000111111	15
b01110001	7
b0111001	8
b011101	25
b011110	4
b011111	26
b100	0
b1010	3
b1011	1
b110	10
b111	9

VP3.1 Huffman Table Number 61

Huffman Code	Token Value
b00	9
b010	3
b01100	23
b011010	27
b011011	26
b0111	2
b100	0
b10100	4
b101010	24
b101011	12
b101100	11
b1011010	25
b101101100	5
b1011011010000	14
b1011011010001	6
b1011011010010	16
b1011011010011	15
b10110110101	31
b1011011011000	22
b1011011011001	21
b10110110110100	18
b10110110110101	17
b10110110110110	20
b10110110110111	19
b10110110111	30
b101101110	8
b1011011110	13
b1011011111	7
b10111	28
b1100	29
b1101	1
b111	10

VP3.1 Huffman Table Number 62

Huffman Code	Token Value
b00	10
b01	9
b10	0
b1100	3
b1101000	12
b1101001	11
b110101000	24
b110101001	23
b11010101	27
b110101100	5
b1101011010	25
b1101011011000	6
b11010110110010	8
b11010110110011	7
b11010110110100	22
b11010110110101	21
b11010110110110	31
b11010110110111	30
b11010110111000	18
b11010110111001	17
b11010110111010	20
b11010110111011	19
b11010110111100	14
b11010110111101	13
b11010110111110	16
b11010110111111	15
b11010111	26
b11011	29
b11100	2
b111010	28
b111011	4
b1111	1

VP3.1 Huffman Table Number 63

Huffman Code	Token Value
b000	0
b0010	28
b0011	13
b010	9
b011	10
b10000	1
b10001	14
b10010	25
b10011	31
b10100	7
b1010100	16
b101010100	4
b101010101000	6
b1010101010010	20
b1010101010011	19
b1010101010100	22
b1010101010101	21
b101010101011	5
b1010101011	18
b10101011	17
b101011	27
b1011	12
b1100	11
b1101	23
b1110000	8
b1110001	3
b1110010	2
b1110011	15
b11101	30
b11110	24
b111110	26
b111111	29

VP3.1 Huffman Table Number 64

Huffman Code	Token Value
b00000	26
b00001	31
b00010	7
b0001100	4
b000110100000	22
b000110100001	21
b0001101000100	18
b0001101000101	6
b0001101000110	20
b0001101000111	19
b0001101001	5
b0001101010	17
b0001101011	16
b00011011	15
b000111	14
b001	10
b010	9
b011	0
b1000	28
b10010	25
b10011	30
b101000	8
b101001	2
b10101	13
b1011	23
b1100	12
b11010	24
b11011	29
b1110	11
b111100	27
b111101	3
b11111	1

VP3.1 Huffman Table Number 65

Huffman Code	Token Value
b000	9
b001	10
b0100	1
b0101	29
b01100	13
b01101	25
b0111	28
b100	0
b10100	3
b101010	8
b1010110	4
b101011100	5
b1010111010000	20
b1010111010001	19
b1010111010010	22
b1010111010011	21
b1010111010100	16
b1010111010101	6
b1010111010110	18
b1010111010111	17
b1010111011	15
b10101111	14
b1011	23
b110000	31
b110001	27
b11001	24
b1101	12
b1110	11
b111100	7
b111101	30
b111110	26
b111111	2

VP3.1 Huffman Table Number 66

Huffman Code	Token Value
b0000	3
b000100000000	18
b000100000001	17
b000100000010	20
b000100000011	19
b000100000100	22
b000100000101	21
b000100000110	6
b00001000001110	16
b00001000001111	15
b000100001	14
b00010001	5
b0001001	31
b0001010	7
b0001011	30
b00011	25
b0010	12
b0011	11
b010	9
b011	10
b1000	1
b1001	28
b101	0
b1100	23
b11010	2
b110110	4
b1101110	8
b1101111	13
b1110	29
b11110	24
b111110	26
b111111	27

VP3.1 Huffman Table Number 67

Huffman Code	Token Value
b000	29
b00100	25
b0010100	8
b00101010	13
b0010101100	31
b0010101101	6
b0010101110000	18
b0010101110001	17
b0010101110010	20
b0010101110011	19
b0010101110100	22
b0010101110101	21
b0010101110110	14
b00101011101110	16
b00101011101111	15
b0010101111	7
b001011	27
b0011	23
b010	0
b011	10
b100	9
b1010	28
b10110	24
b10111	12
b1100	3
b11010	11
b110110	26
b1101110	5
b1101111	30
b1110	1
b11110	2
b11111	4

VP3.1 Huffman Table Number 68

Huffman Code	Token Value
b000	23
b001	3
b0100	4
b0101	1
b011	2
b100	0
b101000	24
b101001000	26
b1010010010000	17
b1010010010001	16
b1010010010010	19
b1010010010011	18
b1010010010100	13
b1010010010101	7
b1010010010110	15
b1010010010111	14
b10100100110	6
b101001001110	30
b10100100111100	21
b10100100111101	20
b10100100111110	31
b10100100111111	22
b10100101	25
b10100110	8
b10100111	27
b10101	29
b101100	12
b101101	11
b101110	28
b101111	5
b110	10
b111	9

VP3.1 Huffman Table Number 69

Huffman Code	Token Value
b000	23
b001	3
b0100	4
b0101	1
b011	2
b100	0
b101000	24
b101001000	26
b1010010010000	17
b1010010010001	16
b1010010010010	19
b1010010010011	18
b1010010010100	13
b1010010010101	7
b1010010010110	15
b1010010010111	14
b10100100110	6
b101001001110	30
b101001001111100	21
b101001001111101	20
b101001001111110	31
b101001001111111	22
b10100101	25
b10100110	8
b10100111	27
b10101	29
b101100	12
b101101	11
b101110	28
b101111	5
b110	10
b111	9

VP3.1 Huffman Table Number 70

Huffman Code	Token Value
b000	23
b001	3
b0100	4
b0101	1
b011	2
b100	0
b101000	24
b101001000	26
b1010010010000	17
b1010010010001	16
b1010010010010	19
b1010010010011	18
b1010010010100	13
b1010010010101	7
b1010010010110	15
b1010010010111	14
b10100100110	6
b101001001110	30
b101001001111100	21
b101001001111101	20
b101001001111110	31
b101001001111111	22
b10100101	25
b10100110	8
b10100111	27
b10101	29
b101100	12
b101101	11
b101110	28
b101111	5
b110	10
b111	9

VP3.1 Huffman Table Number 71

Huffman Code	Token Value
b000	10
b001	9
b0100	24
b01010	7
b01011	26
b011	0
b100000	2
b1000010	15
b100001100000	6
b1000011000010	20
b1000011000011	19
b1000011000100	22
b1000011000101	21
b100001100011	5
b1000011001	18
b100001101	4
b100001110	17
b100001111	16
b10001	1
b1001	28
b1010	12
b1011	11
b11000	13
b11001	25
b11010	30
b11011	29
b111000	14
b111001	27
b1110100	3
b1110101	8
b111011	31
b1111	23

VP3.1 Huffman Table Number 72

Huffman Code	Token Value
b00000	13
b00001	3
b0001	1
b001	10
b010	9
b0110	29
b01110	25
b011110	31
b011111	8
b1000	12
b1001	11
b101	0
b1100	28
b1101	23
b1110000	14
b11100010	4
b1110001100	16
b11100011010	17
b11100011011000	18
b11100011011001	6
b11100011011010	20
b11100011011011	19
b11100011011100	22
b11100011011101	21
b1110001101111	5
b111000111	15
b111001	7
b11101	24
b111100	27
b111101	30
b111110	2
b111111	26

VP3.1 Huffman Table Number 73

Huffman Code	Token Value
b000000	31
b000001	7
b00001	25
b0001	28
b001	9
b010	10
b0110	12
b0111	11
b100000	30
b100001	8
b10001	2
b1001	29
b1010	23
b1011	1
b110	0
b11100	24
b1110100	4
b111010100	15
b1110101010	5
b11101010011000	20
b11101010011001	19
b11101010011010	22
b11101010011011	21
b11101010011100	6
b11101010111010	18
b11101010111011	17
b111010101111	16
b11101011	14
b111011	27
b11110	3
b111110	13
b111111	26

VP3.1 Huffman Table Number 74

Huffman Code	Token Value
b0000	12
b0001	11
b00100	25
b001010	13
b0010110	30
b0010111	7
b0011	28
b0100	3
b01010	24
b010110	4
b010111	27
b0110	23
b0111	29
b100	0
b1010	1
b101100	26
b10110100	31
b101101010	5
b1011010110000	16
b1011010110001	6
b1011010110010	18
b1011010110011	17
b101101011010	15
b10110101101100	20
b10110101101101	19
b10110101101110	22
b10110101101111	21
b1011010111	14
b1011011	8
b10111	2
b110	9
b111	10

VP3.1 Huffman Table Number 75

Huffman Code	Token Value
b00	9
b0100	28
b0101	2
b01100000	30
b01100001	7
b0110001	8
b011001	27
b011010	24
b011011	25
b0111	29
b10000	11
b10001	12
b1001	3
b101	0
b11000	23
b11001000	13
b1100100100	31
b1100100101000	18
b1100100101001	17
b1100100101010	20
b1100100101011	19
b1100100101100	6
b11001001011010	16
b11001001011011	15
b11001001011100	22
b11001001011101	21
b1100100101111	14
b110010011	5
b1100101	26
b110011	4
b1101	1
b111	10

VP3.1 Huffman Table Number 76

Huffman Code	Token Value
b000	1
b0010	2
b00110	29
b001110	12
b001111	11
b01	9
b10	10
b110	0
b111000	23
b111001	4
b111010	28
b111011000000	30
b1110110000010	6
b11101100000110	15
b11101100000111	14
b11101100001	7
b11101100010	13
b11101100011000	21
b11101100011001	20
b11101100011010	31
b11101100011011	22
b11101100011100	17
b11101100011101	16
b11101100011110	19
b11101100011111	18
b111011001	5
b11101101	25
b11101110	27
b111011110	24
b1110111110	8
b1110111111	26
b1111	3

VP3.1 Huffman Table Number 77

Huffman Code	Token Value
b00	0
b010	1
b0110	3
b011100	4
b0111010000	5
b0111010001000	14
b0111010001001	13
b0111010001010	16
b0111010001011	15
b0111010001100	6
b01110100011010	8
b01110100011011	7
b01110100011100	27
b01110100011101	26
b01110100011110	31
b01110100011111	30
b011101001	12
b011101010	11
b01110101100000	22
b01110101100001	21
b01110101100010	25
b01110101100011	24
b01110101100100	18
b01110101100101	17
b01110101100110	20
b01110101100111	19
b01110101101	23
b0111010111	29
b0111011	28
b01111	2
b10	10
b11	9

VP3.1 Huffman Table Number 78

Huffman Code	Token Value
b00	10
b01	9
b10	0
b1100	3
b1101000	12
b1101001	11
b110101000	24
b110101001	23
b11010101	27
b110101100	5
b1101011010	25
b1101011011000	6
b11010110110010	8
b11010110110011	7
b11010110110100	22
b11010110110101	21
b11010110110110	31
b11010110110111	30
b11010110111000	18
b11010110111001	17
b11010110111010	20
b11010110111011	19
b11010110111100	14
b11010110111101	13
b11010110111110	16
b11010110111111	15
b11010111	26
b11011	29
b11100	2
b111010	28
b111011	4
b1111	1

VP3.1 Huffman Table Number 79

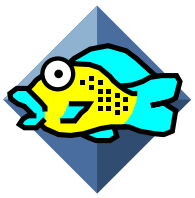
Appendix C

Colophon

Ogg is a Xiph.org Foundation effort to protect essential tenets of Internet multimedia from corporate hostage-taking; Open Source is the net's greatest tool to keep everyone honest. See About the Xiph.org Foundation for details.

Ogg Theora is the first Ogg video codec. Anyone may freely use and distribute the Ogg and Theora specifications, whether in private, public, or corporate capacity. However, the Xiph.org Foundation and the Ogg project reserve the right to set the Ogg Theora specification and certify specification compliance.

Xiph.org's Theora software codec implementation is distributed under a BSD-like license. This does not restrict third parties from distributing independent implementations of Theora software under other licenses.



This document is set in LaTeX.

Bibliography

[Bra97] Scott Bradner. *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels*, March 1997. http://www.ietf.org/rfc/rfc2119.txt.

[CSF77] Wen-Hsiung Chen, C. Harrison Smith, and S. C. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on Communications*, COM-25(9):1004–1011, September 1977.

[ITU95] International Telecommunications Union, 1211 Geneva 20, Switzerland. *Reccomendation ITU-R BT.601-5: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios*, 1982, revised edition, 1995.

[ITU98] International Telecommunications Union, 1211 Geneva 20, Switzerland. *Reccomendation ITU-R BT.470-6: Conventional Television Systems*, 1970, revised edition, 1998.

[ITU02] International Telecommunications Union, 1211 Geneva 20, Switzerland. *Recommendation ITU-R BT.709-5: Parameter values for the HDTV standards for production and international programme exchange*, 1990, revised edition, 2002.

[Mel04] Mike Melanson. VP3 bitstream format and decoding process. http://www.multimedia.cx/vp3-format.txt, March 2004.

[Pfe03] Silvia Pfeiffer. *RFC 3533: The Ogg Encapsulation Format Version 0*, May 2003. http://www.ietf.org/rfc/rfc3533.txt.

[Poy97] Charles Poynton. Frequently-asked questions about gamma. http://www.poynton.com/GammaFAQ.html, February 1997.

[SMP94] Society of Motion Pciture and Television Engineers. *SMPTE-170M: Television — Composite Analog Video Signal — NTSC for Studio Applications*, 1994.

[Xip02] Xiph.org Foundation. *Vorbis I specification*, 2002. http://www.xiph.org/ogg/vorbis/doc/.

[Yer96] Francois Yergeau. *RFC 2044: UTF-8, a transformation format of Unicode and ISO 10646*, October 1996. http://www.ietf.org/rfc/rfc2044.txt.