VP8 Data Format and Decoding Specification

WebM Project

Google, Inc.
3 Corporate Drive, Suite 100
Clifton Park, NY 12065 USA
http://www.webmproject.org/

Revised: May 2010

License
Copyright
Chapter 1: Introduction
Chapter 2: Format Overview
Chapter 3: Compressed Frame Types
Chapter 4: Overview of Compressed Data Format
Chapter 5: Overview of the Decoding Process
Chapter 6: Description of Algorithms
Chapter 7: Boolean Entropy Decoder

7.1 Underlying Theory of Coding
7.2 Practical Algorithm Description
7.3 Actual Implementation

Chapter 8: Compressed Data Components

8.1 Tree Coding Implementation
8.2 Tree Coding Example

Chapter 9: Frame Header

9.1 Uncompressed Data Chunk
9.2 Color Space and Pixel Type (Key Frames-only)
9.3 Segment-based Adjustments
9.4 Loop Filter Type and Levels
9.5 Token Partition and Partition Data Offsets
9.6 Dequantization Indices
9.7 Refresh Golden Frame and AltRef Frame
9.8 Refresh Last Frame Buffer
9.9 DCT Coefficient Probability Update
9.10 Remaining Frame Header Data (non-Key Frame)
9.11 Remaining Frame Header Data (Key Frame)

Chapter 10: Segment-based Feature Adjustments
Chapter 11: Key Frame Macroblock Prediction Records

11.1 mb_skip_coeff
11.2 Luma Modes
11.3 Subblock Mode Contexts
11.4 Chroma Modes
11.5 Subblock Mode Probability Table

Chapter 12: Intraframe Prediction

12.1 mb_skip_coeff
12.2 Chroma Prediction
12.3 Luma Prediction

Chapter 13: DCT Coefficient Decoding

13.1 MB Without non-Zero Coefficient Values
13.2 Coding of Individual Coefficient Values
13.3 Token Probabilities

13.4 Token Probability Updates
13.5 Default Token Probability Table

Chapter 14: DCT and WHT Inversion and Macroblock Reconstruction

14.1 Dequantization
14.2 Inverse Transforms
14.3 Implementation of the WHT Inversion
14.4 Implementation of the DCT Inversion
14.5 Summation of Predictor and Residue

Chapter 15: Loop Filter

15.1 Filter Geometry and Overall Procedure
15.2 Simple Filter
15.3 Normal Filter
15.4 Calculation of Control Parameters

Chapter 16: Interframe Macroblock Prediction Records

16.1 Intra-Predicted Macroblocks
16.2 Inter-Predicted Macroblocks
16.3 Mode and Motion Vector Contexts
16.4 Split Prediction

Chapter 17: Motion Vector Decoding

17.1 Coding of Each Component
17.2 Probability Updates

Chapter 18: Interframe Prediction

18.1 Bounds on and Adjustment of Motion Vectors
18.2 Prediction Subblocks
18.3 Sub-pixel Interpolation
18.4 Filter Properties

Chapter 19: References

Revision History

License

Google hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise implementations of this specification where such license applies only to those patent claims, both currently owned by Google and acquired in the future, licensable by Google that are necessarily infringed by implementation of this specification. If You or your agent or exclusive licensee institute or order or agree to the institution of patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that any implementation of this specification constitutes direct or contributory patent infringement, or inducement of patent infringement, then any rights granted to You under the License for this specification shall terminate as of the date such litigation is filed.

Copyright

Chapter 1: Introduction

This document describes the VP8 compressed video data format created by Google On2, together with a discussion of the decoding procedure for this format. It is intended to be used in conjunction with and as a guide to the reference decoder provided by Google On2. If there are any conflicts between this document and the reference source code, the reference source code should be considered correct. The bitstream is defined by the reference source code and not this document.

Like many modern video compression schemes, VP8 is based on decomposition of frames into square subblocks of pixels, prediction of such subblocks using previously constructed blocks, and adjustment of such predictions (as well as synthesis of unpredicted blocks) using a discrete cosine transform (hereafter abbreviated as DCT). In one special case, however, VP8 uses a “Walsh-Hadamard” (hereafter abbreviated as WHT) transform instead of a DCT.

Roughly speaking, such systems reduce datarate by exploiting the temporal and spatial coherence of most video signals. It is more efficient to specify the location of a visually similar portion of a prior frame than it is to specify pixel values. The frequency segregation provided by the DCT and WHT facilitate the exploitation of both spatial coherence in the original signal and the tolerance of the human visual system to moderate losses of fidelity in the reconstituted signal.

VP8 augments these basic concepts with, among other things, sophisticated usage of contextual probabilities. The result is a significant reduction in datarate at a given quality.

Unlike some similar schemes (the older MPEG formats, for example), VP8 specifies exact values for reconstructed pixels. Specifically, the specification for the DCT and WHT portions of the reconstruction does not allow for any “drift” caused by truncation of fractions. Rather, the algorithm is specified using fixed-precision integer operations exclusively. This greatly facilitates the verification of the correctness of a decoder implementation as well as avoiding difficult-to-predict visual incongruities between such implementations.

It should be remarked that, in a complete video playback system, the displayed frames may or may not be identical to the reconstructed frames. Many systems apply a final level of filtering (commonly referred to as postprocessing) to the reconstructed frames prior to viewing. Such postprocessing has no effect on the decoding and reconstruction of subsequent frames (which are predicted using the completely-specified reconstructed frames) and is beyond the scope of this document. In practice, the nature and extent of this sort of postprocessing is dependent on both the taste of the user and on the computational facilities of the playback environment.

Chapter 2: Format Overview

VP8 works exclusively with an 8-bit YUV 4:2:0 image format. In this format, each 8-bit pixel in the two chroma planes (U and V) corresponds positionally to a 2x2 block of 8-bit luma pixels in the Y plane; coordinates of the upper left corner of the Y block are of course exactly twice the coordinates of the corresponding chroma pixels. When we refer to pixels or pixel distances without specifying a plane, we are implicitly referring to the Y plane or to the complete image, both of which have the same (full) resolution.

As is usually the case, the pixels are simply a large array of bytes stored in rows from top to bottom, each row being stored from left to right. This “left to right” then “top to bottom” raster-scan order is reflected in the layout of the compressed data as well.

Provision has been made for the support of two different YUV color formats in the VP8 bitstream header, however only one format is supported in the first release of VP8.

The YUV formats differ in terms of their conversion to and from RGB color space. The first corresponds to the traditional YUV color space similar to the YCrCb color space defined in ITU-R BT.601. The second (currently unsupported) format corresponds to a new YUV color space whose digital conversion to and from RGB can be implemented without multiplications and divides. The VP8 Decoder should decode and pass the information on to the processes that convert the YUV output to RGB color space.

Occasionally, at very low datarates, a compression system may decide to reduce the resolution of the input signal to facilitate efficient compression. The VP8 data format supports this via optional upscaling of its internal reconstruction buffer prior to output (this is completely distinct from the optional postprocessing discussed earlier, which has nothing to do with decoding per se). This upsampling restores the video frames to their original resolution. In other words, the compression/decompression system can be viewed as a “black box”, where the input and output is always at a given resolution. The compressor might decide to “cheat” and process the signal at a lower resolution. In that case, the decompressor needs the ability to restore the signal to its original resolution.

Internally, VP8 decomposes each output frame into an array of macroblocks. A macroblock is a square array of pixels whose Y dimensions are 16x16 and whose U and V dimensions are 8x8. Macroblock-level data in a compressed frame occurs (and must be processed) in a raster order similar to that of the pixels comprising the frame.

Macroblocks are further decomposed into 4x4 subblocks. Every macroblock has 16 Y subblocks, 4 U subblocks, and 4 V subblocks. Any subblock-level data (and processing of such data) again occurs in raster order, this time in raster order within the containing macroblock.

As discussed in further detail below, data can be specified at the levels of both macroblocks and their subblocks.

Pixels are always treated, at a minimum, at the level of subblocks, which may be thought of as the “atoms” of the VP8 algorithm. In particular, the 2x2 chroma blocks corresponding to 4x4 Y subblocks are never treated explicitly in the data format or in the algorithm specification.

The DCT and WHT always operate at a 4x4 resolution. The DCT is used for the 16Y, 4U and 4V subblocks. The WHT is used (with some but not all prediction modes) to encode a 4x4 array comprising the average intensities of the 16 Y subblocks of a macroblock. These average intensities are, up to a constant

normalization factor, nothing more that the zeroth DCT coefficients of the Y subblocks. This “higher-level” WHT is a substitute for the explicit specification of those coefficients, in exactly the same way as the DCT of a subblock substitutes for the specification of the pixel values comprising the subblock. We consider this 4x4 array as a *second-order* subblock called Y2, and think of a macroblock as containing 24 “real” subblocks and, sometimes, a 25th “virtual” subblock. This is dealt with further in Chapter 13.

The frame layout used by the reference decoder may be found in the file `yv12config.h`.

Chapter 3: Compressed Frame Types

There are only two types of frames in VP8.

Intraframes (also called *key frames* and, in MPEG terminology, *I-frames*) are decoded without reference to any other frame in a sequence, that is, the decompressor reconstructs such frames beginning from its "default" state. Key frames provide random access (or seeking) points in a video stream.

Interframes (also called *prediction frames* and, in MPEG terminology, *P-frames*) are encoded with reference to prior frames, specifically all prior frames up to and including the most recent key frame. Generally speaking, the correct decoding of an interframe depends on the correct decoding of the most recent key frame and all ensuing frames. Consequently, the decoding algorithm is not tolerant of dropped frames: In an environment in which frames may be dropped or corrupted, correct decoding will not be possible until a key frame is correctly received.

In contrast to MPEG, there is no use of bidirectional prediction. No frame is predicted using frames temporally subsequent to it; there is no analog to an MPEG B-frame.

Secondly, VP8 augments these notions with that of alternate prediction frames, called *golden frames* and *altref frames* (alternative reference frames). Blocks in an interframe may be predicted using blocks in the immediately previous frame as well as the most recent golden frame or altref frame. Every key frame is automatically golden and altref, and any interframe may optionally replace the most recent golden or altref frame.

Golden frames and altref frames may also be used to partially overcome the intolerance to dropped frames discussed above: If a compressor is configured to code golden frames only with reference to the prior golden frame (and key frame) then the "substream" of key and golden frames may be decoded regardless of loss of other interframes. Roughly speaking, the implementation requires (on the compressor side) that golden frames subsume and recode any context updates effected by the intervening interframes. A typical application of this approach is video conferencing, in which retransmission of a prior golden frame and/or a delay in playback until receipt of the next golden frame is preferable to a larger retransmit and/or delay until the next key frame.

Chapter 4: Overview of Compressed Data Format

The input to a VP8 decoder is a sequence of compressed frames whose order matches their order in time. Issues such as the duration of frames, the corresponding audio, and synchronization are generally provided by the playback environment and are irrelevant to the decoding process itself, however, to aid in fast seeking a start code is included in the header of each key frame.

The decoder is simply presented with a sequence of compressed frames and produces a sequence of decompressed (reconstructed) YUV frames corresponding to the input sequence. As stated in the introduction, the exact pixel values in the reconstructed frame are part of VP8's specification. This document specifies the layout of the compressed frames and gives unambiguous algorithms for the correct production of reconstructed frames.

The first frame presented to the decompressor is of course a key frame. This may be followed by any number of interframes; the correct reconstruction of each frame depends on all prior frames up to the key frame. The next key frame restarts this process: The decompressor resets to its default initial condition upon reception of a key frame and the decoding of a key frame (and its ensuing interframes) is completely independent of any prior decoding.

At the highest level, every compressed frame has three or more pieces. It begins with an uncompressed data chunk comprising 10 bytes in the case of key frames and 3-bytes for inter frames. This is followed by two or more blocks of compressed data (called *partitions*). These compressed data partitions begin and end on byte boundaries.

The first compressed partition has two subsections:

1. Header information that applies to the frame as a whole.
2. Per-macroblock information specifying how each macroblock is predicted from the already-reconstructed data that is available to the decompressor.

As stated above, the macroblock-level information occurs in raster-scan order.

The rest of the partitions contain, for each block, the DCT/WHT coefficients (quantized and logically compressed) of the residue signal to be added to the predicted block values. It typically accounts for roughly 70% of the overall datarate. VP8 supports packing the compressed DCT/WHT coefficients' data from macroblock rows into separate partitions. If there is more than one partition for these coefficients, the sizes of the partitions — except the last partition — in bytes are also present in the bitstream right after the above first partition. Each of the sizes is a 3-byte data item written in little endian format. These sizes provide the decoder direct access to all DCT/WHT coefficient partitions, which enables parallel processing of the coefficients in a decoder.

The separate partitioning of the prediction data and coefficient data also allows flexibility in the implementation of a decompressor: An implementation may decode and store the prediction information for the whole frame and then decode, transform, and add the residue signal to the entire frame, or it may simultaneously decode both partitions, calculating prediction information and adding in the residue signal for each block in order. The length field in the frame tag, which allows decoding of the second partition to begin before the first partition has been completely decoded, is necessary for the second "block-at-a-time" decoder implementation.

All partitions are decoded using separate instances of the boolean entropy decoder described in Chapter 7. Although some of the data represented within the partitions is conceptually “flat” (a bit is just a bit with no probabilistic expectation one way or the other), because of the way such coders work, there is never a direct correspondence between a “conceptual bit” and an actual physical bit in the compressed data partitions. Only in the 3 or 10 byte uncompressed chunk described above is there such a physical correspondence.

A related matter, which is true for most lossless compression formats, is that seeking within a partition is not supported. The data must be decompressed and processed (or at least stored) in the order in which it occurs in the partition.

While this document specifies the ordering of the partition data correctly, the details and semantics of this data are discussed in a more logical fashion to facilitate comprehension. For example, the frame header contains updates to many probability tables used in decoding per-macroblock data. The latter is often described before the layouts of the probabilities and their updates, even though this is the opposite of their order in the bitstream.

Chapter 5: Overview of the Decoding Process

A VP8 decoder needs to maintain four YUV frame buffers whose resolutions are at least equal to that of the encoded image. These buffers hold the current frame being reconstructed, the immediately previous reconstructed frame, the most recent golden frame, and the most recent altref frame.

Most implementations will wish to “pad” these buffers with “invisible” pixels that extend a moderate number of pixels beyond all four edges of the visible image. This simplifies interframe prediction by allowing all (or most) prediction blocks — which are *not* guaranteed to lie within the visible area of a prior frame — to address usable image data.

Regardless of the amount of padding chosen, the invisible rows above (below) the image are filled with copies of the top (bottom) row of the image; the invisible columns to the left (right) of the image are filled with copies of the leftmost (rightmost) visible row; and the four invisible corners are filled with copies of the corresponding visible corner pixels. The use of these prediction buffers (and suggested sizes for the *halo*) will be elaborated on in the discussion of motion vectors, interframe prediction, and sub-pixel interpolation later in this document.

As will be seen in the description of the frame header, the image dimensions are specified (and can change) with every key frame. These buffers (and any other data structures whose size depends on the size of the image) should be allocated (or re-allocated) immediately after the dimensions are decoded.

Leaving most of the details for later elaboration, the following is an outline the decoding process.

First, the frame header (beginning of the first data partition) is decoded. Altering or augmenting the maintained state of the decoder, this provides the context in which the per-macroblock data can be interpreted.

The macroblock data occurs (and must be processed) in raster-scan order. This data comes in two or more parts. The first (*prediction* or *mode*) part comes in the remainder of the first data partition. The other parts comprise the data partition(s) for the DCT/WHT coefficients of the residue signal. For each macroblock, the prediction data must be processed before the residue.

Each macroblock is predicted using one (and only one) of four possible frames. All macroblocks in a key frame, and all *intra-coded* macroblocks in an interframe, are predicted using the already-decoded macroblocks in the current frame. Macroblocks in an interframe may also be predicted using the previous frame, the golden frame or the altref frame. Such macroblocks are said to be *inter-coded*.

The purpose of prediction is to use already-constructed image data to approximate the portion of the original image being reconstructed. The effect of any of the prediction modes is then to write a macroblock-sized prediction buffer containing this approximation.

Regardless of the prediction method, the residue DCT signal is decoded, dequantized, reverse-transformed, and added to the prediction buffer to produce the (almost final) reconstruction value of the macroblock, which is stored in the correct position of the current frame buffer.

The residue signal consists of 24 (sixteen Y, four U, and four V) 4x4 quantized and losslessly-compressed DCT transforms approximating the difference between the original macroblock in the uncompressed source

and the prediction buffer. For most prediction modes, the zeroth coefficients of the sixteen Y subblocks are expressed via a 25th WHT of the second-order virtual Y2 subblock discussed above.

Intra-prediction exploits the spatial coherence of frames. The 16x16 luma (Y) and 8x8 chroma (UV) components are predicted independently of each other using one of four simple means of pixel propagation, starting from the already-reconstructed (16-pixel long luma, 8-pixel long chroma) row above and column to the left of the current macroblock. The four methods are:

1. Copying the row from above throughout the prediction buffer.
2. Copying the column from left throughout the prediction buffer.
3. Copying the average value of the row and column throughout the prediction buffer.
4. Extrapolation from the row and column using the (fixed) second difference (horizontal and vertical) from the upper left corner.

Additionally, the sixteen Y subblocks may be predicted independently of each other using one of ten different *modes*, four of which are 4x4 analogs of those described above, augmented with six “diagonal” prediction methods. There are two types of predictions, one intra and one prediction (among all the modes), for which the residue signal does not use the Y2 block to encode the DC portion of the sixteen 4x4 Y subblock DCTs. This “independent Y subblock” mode has no effect on the 8x8 chroma prediction.

Inter-prediction exploits the temporal coherence between nearby frames. Except for the choice of the prediction frame itself, there is no difference between inter-prediction based on the previous frame and that based on the golden frame or altref frame.

Inter-prediction is conceptually very simple. While, for reasons of efficiency, there are several methods of encoding the relationship between the current macroblock and corresponding sections of the prediction frame, ultimately each of the sixteen Y subblocks is related to a 4x4 subblock of the prediction frame, whose position in that frame differs from the current subblock position by a (usually small) displacement. These two-dimensional displacements are called *motion vectors*.

The motion vectors used by VP8 have quarter-pixel precision. Prediction of a subblock using a motion vector that happens to have integer (whole number) components is very easy: the 4x4 block of pixels from the displaced block in the previous, golden, or altref frame are simply copied into the correct position of the current macroblock’s prediction buffer.

Fractional displacements are conceptually and implementationally more complex. They require the inference (or synthesis) of sample values that, strictly speaking, do not exist. This is one of the most basic problems in signal processing and readers conversant with that subject will see that the approach taken by VP8 provides a good balance of robustness, accuracy, and efficiency.

Leaving the details for the implementation discussion below, the pixel interpolation is calculated by applying a kernel filter (using reasonable-precision integer math) three pixels on either side, both horizontally and vertically, of the pixel to be synthesized. The resulting 4x4 block of synthetic pixels is then copied into position exactly as in the case of integer displacements.

Each of the eight chroma subblocks is handled similarly. Their motion vectors are never specified explicitly; instead, the motion vector for each chroma subblock is calculated by averaging the vectors of the four Y subblocks that occupy the same area of the frame. Since chroma pixels have twice the diameter (and four times the area) of luma pixels, the calculated chroma motion vectors have 1/8 pixel resolution, but the

procedure for copying or generating pixels for each subblock is essentially identical to that done in the luma plane.

After all the macroblocks have been generated (predicted and corrected with the DCT/WHT residue), a filtering step (the *loop filter*) is applied to the entire frame. The purpose of the loop filter is to reduce blocking artifacts at the boundaries between macroblocks and between subblocks of the macroblocks. The term loop filter is used because this filter is part of the "coding loop," that is, it affects the reconstructed frame buffers that are used to predict ensuing frames. This is distinguished from the postprocessing filters discussed earlier which affect only the viewed video and do not "feed into" subsequent frames.

Next, if signaled in the data, the current frame (or individual macroblocks within the current frame) may replace the golden frame prediction buffer and/or the altref frame buffer.

The halos of the frame buffers are next filled as specified above. Finally, at least as far as decoding is concerned, the (references to) the "current" and "last" frame buffers should be exchanged in preparation for the next frame.

Various processes may be required (or desired) before viewing the generated frame. As discussed in the frame dimension information below, truncation and/or upscaling of the frame may be required. Some playback systems may require a different frame format (RGB, YUY2, etc.). Finally, as mentioned in the introduction, further postprocessing or filtering of the image prior to viewing may be desired. Since the primary purpose of this document is a decoding specification, the postprocessing is not specified in this document.

While the basic ideas of prediction and correction used by VP8 are straightforward, many of the details are quite complex. The management of probabilities is particularly elaborate. Not only do the various modes of intra-prediction and motion vector specification have associated probabilities but they, together with the coding of DCT coefficients and motion vectors, often base these probabilities on a variety of contextual information (calculated from what has been decoded so far), as well as on explicit modification via the frame header.

The "top-level" of decoding and frame reconstruction is implemented in the reference decoder files `onyxd_if.c` and `decodframe.c`.

This concludes our summary of decoding and reconstruction; we continue by discussing the individual aspects in more depth.

A reasonable "divide and conquer" approach to implementation of a decoder is to begin by decoding streams composed exclusively of key frames. After that works reliably, interframe handling can be added more easily than if complete functionality were attempted immediately. In accordance with this, we first discuss components needed to decode key frames (most of which are also used in the decoding of interframes) and conclude with topics exclusive to interframes.

Chapter 6: Description of Algorithms

As the intent of this document, together with the reference decoder source code, is to specify a platform-independent procedure for the decoding and reconstruction of a VP8 video stream, many (small) algorithms must be described exactly.

Due to its near-universality, terseness, ability to easily describe calculation at specific precisions, and the fact that On2's reference VP8 decoder is written in C, these algorithm fragments are written using the C programming language, augmented with a few simple definitions below.

The standard (and best) reference for C is *The C Programming Language*, written by Brian W. Kernighan and Dennis M. Ritchie, and published by Prentice-Hall.

Many code fragments will be presented in this document. Some will be nearly identical to corresponding sections of the reference decoder; others will differ. Roughly speaking, there are three reasons for such differences:

1. For reasons of efficiency, the reference decoder version may be less obvious.
2. The reference decoder often uses large data structures to maintain context that need not be described or used here.
3. The authors of this document felt that a different expression of the same algorithm might facilitate exposition.

Regardless of the chosen presentation, the calculation effected by any of the algorithms described here is identical to that effected by the corresponding portion of the reference decoder.

All VP8 decoding algorithms use integer math. To facilitate specification of arithmetic precision, we define the following types.

```
typedef   signed char  int8;    /* signed integer exactly 8 bits wide */
typedef unsigned char uint8;    /* unsigned "" */

typedef short int16;            /* signed integer exactly 16 bits wide */
typedef unsigned int16 uint16;  /* unsigned "" */

/* int32 is a signed integer type at least 32 bits wide */

typedef long int32; /* guaranteed to work on all systems */
typedef int  int32; /* will be more efficient on some systems */

typedef unsigned int32 uint32;

/* unsigned integer type, at least 16 bits wide, whose exact size
   is most convenient to whatever processor we are using */

typedef unsigned int uint;

/* While pixels themselves are 8-bit unsigned integers,
   pixel arithmetic often occurs at 16- or 32-bit precision and
   the results need to be "saturated" or clamped to an 8-bit range. */
```

```
typedef uint8 Pixel;

Pixel clamp255( int32 v) { return v < 0? 0 : (v < 255? v : 255);}

/*  As is elaborated in the discussion of the bool_decoder below, VP8
represents probabilities as unsigned 8-bit numbers. */

typedef uint8 Prob;
```

We occasionally need to discuss mathematical functions involving honest-to-goodness “infinite precision” real numbers. The DCT is first described via the cosine function $\cos$; the ratio of the lengths of the circumference and diameter of a circle is denoted π; at one point, we take a (base $1/2$) logarithm denoted $\log$; and $pow(x, y)$ denotes x raised to the power y. If $x = 2$ and y is a small non-negative integer, $pow(2, y)$ may be expressed in C as `1 << y` .

Finally, we sometimes need to divide signed integers by powers of two, that is, we occasionally right-shift signed numbers. The behavior of such shifts (i.e., the propagation of the sign bit) is, perhaps surprisingly, not defined by the C language itself and is left up to individual compilers. Because of the utility of this frequently needed operation, it is at least arguable that it should be defined by the language (to naturally propagate the sign bit) and, at a minimum, should be correctly implemented by any reasonable compiler. In the interest of strict portability, we attempt to call attention to these shifts when they arise.

Chapter 7: Boolean Entropy Decoder

As discussed in the overview above, essentially the entire VP8 data stream is encoded using a boolean entropy coder.

An understanding of the `bool_decoder` is critical to the implementation of a VP8 decompressor, so we discuss in detail. It is easier to comprehend the `bool_decoder` in conjunction with the `bool_encoder` used by the compressor to write the compressed data partitions.

The `bool_encoder` encodes (and the `bool_decoder` decodes) one bool (zero-or-one boolean value) at a time. Its purpose is to losslessly compress a sequence of bools for which the probability of their being zero or one can be well-estimated (via constant or previously-coded information) at the time they are written, using identical corresponding probabilities at the time they are read.

As the reader is probably aware, if a bool is much more likely to be zero than one (for instance), it can, on average, be faithfully encoded using much less than one bit per value. The `bool_encoder` exploits this.

In the 1940s, Claude Shannon proved that there is a lower bound for the average datarate of a faithful encoding of a sequence of bools (whose probability distributions are known and are independent of each other) and also that there are encoding algorithms that approximate this lower bound as closely as one wishes.

If we encode a sequence of bools whose probability of being zero is p (and whose probability of being 1 is $1-p$), the lowest possible datarate per value is

$$p \cdot \log(p) + (1-p) \cdot \log(1-p);$$

taking the logarithms to the base $1/2$ expresses the datarate in bits/value.

We give two simple examples. At one extreme, if $p = 1/2$, then $\log(p) = \log(1-p) = 1$ and the lowest possible datarate per bool is $1/2 + 1/2 = 1$, that is, we cannot do any better than simply literally writing out bits. At another extreme, if p is very small, say $p = 1/1024$, then $\log(p) = 10$, $\log(1-p)$ is roughly .0014, and the lowest possible datarate is approximately $10/1024 + .0014$, roughly 1/100 of a bit per bool.

Because most of the bools in the VP8 datastream have zero-probabilities nowhere near $1/2$, the compression provided by the `bool_encoder` is critical to the performance of VP8.

The bool coder used by VP8 is a variant of an arithmetic coder. An excellent discussion of arithmetic coding (and other lossless compression techniques) can be found in the book *Text Compression* by Timothy C. Bell, John G. Cleary, and Ian H. Witten, published in 1990 by Prentice-Hall.

7.1 Underlying Theory of Coding

The basic idea used by the bool coder is to consider the entire data stream (either of the partitions in our case) as the binary expansion of a single number x with $0 \le x < 1$. The bits (or bytes) in x are of course written from high to low order and if $b[j](B[j])$ is the j^{th} bit (byte) in the partition, the value x is simply the sum (starting with $j = 1$) of $pow(2, -j) \cdot b[j]$ or $pow(256, -j) \cdot B[j]$.

Before the first bool is coded, all values of x are possible.

The coding of each bool restricts the possible values of x in proportion to the probability of what is coded. If $p1$ is the probability of the first bool being zero and a zero is coded, the range of possible x is restricted to $0 \le x < p1$. If a one is coded, the range becomes $p1 \le x < 1$.

The coding continues by repeating the same idea. At every stage, there is an interval $a \le x < b$ of possible values of x. If p is the probability of a zero being coded at this stage and a zero is coded, the interval becomes $a \le x < a + (p \cdot (b - a))$. If a one is coded, the possible x are restricted to $a + (p \cdot (b - a)) \le x < b$.

Assuming only finitely many values are to be coded, after the encoder has received the last bool, it can write as its output any value x that lies in the final interval. VP8 simply writes the left endpoint of the final interval. Consequently, the output it would make if encoding were to stop at any time either increases or stays the same as each bool is encoded.

Decoding parallels encoding. The decoder is presented with the number x, which has only the initial restriction $0 \le x < 1$. To decode the first bool, the decoder is given the first probability $p1$. If $x < p1$, a zero is decoded; if $x \ge p1$, a one is decoded. In either case, the new restriction on x, that is, the interval of possible x, is remembered.

Decoding continues in exactly the same way: If $a \le x < b$ is the current interval and we are to decode a bool with zero-probability p, we return a zero if $a \le x < a + (p \cdot (b - a))$ and a one if $a + (p \cdot (b - a)) \le x < b$. In either case, the new restriction is remembered in preparation for decoding the next bool.

The process outlined above uses real numbers of infinite precision to express the probabilities and ranges. It is true that, if one could actualize this process and coded a large number of bools whose supplied probabilities matched their value distributions, the datarate achieved would approach the theoretical minimum as the number of bools encoded increased.

Unfortunately, computers operate at finite precision and an approximation to the theoretically perfect process described above is necessary. Such approximation increases the datarate but, at quite moderate precision and for a wide variety of data sets, this increase is negligible.

The only conceptual limitations are, first, that coder probabilities must be expressed at finite precision and, second, that the decoder be able to detect each individual modification to the value interval via examination of a fixed amount of input. As a practical matter, many of the implementation details stem from the fact that the coder can function using only a small “window” to incrementally read or write the arbitrarily precise number x.

7.2 Practical Algorithm Description

VP8’s bool coder works with 8-bit probabilities p. The range of such p is $0 \le p \le 255$; the actual probability represented by p is $p / 256$. Also, the coder is designed so that decoding of a bool requires no more than an 8-bit comparison and so that the state of both the encoder and decoder can be easily represented using a small number of unsigned 16-bit integers.

The details are most easily understood if we first describe the algorithm using bit-at-a-time input and output. Aside from the ability to maintain a position in this bitstream and write/read bits, the encoder also needs the ability to add 1 to the bits already output; after writing n bits, adding 1 to the existing output is the same thing as adding $pow(2, -n)$ to x.

Together with the bit position, the encoder must maintain two unsigned 8-bit numbers which we call bottom and range. Writing w for the n bits already written and $S = pow(2, -n - 8)$ for the scale of the current bit

position one byte out, we have the following constraint on all future values v of w (including the final value $v = x$):

$$w + (S \cdot \text{bottom}) \leq v < w + (S \cdot (\text{bottom} + \text{range}))$$

Thus, appending bottom to the already-written bits w gives the left endpoint of the interval of possible values, appending bottom+range gives the right endpoint, range itself (scaled to the current output position) is the length of the interval.

So that our probabilistic encodings are reasonably accurate, we do not let range vary by more than a factor of two: It stays within the bounds $128 \leq \text{range} \leq 255$.

The process for encoding a boolean value val whose probability of being zero is $\frac{\text{prob}}{256}$ — and whose probability of being one is $\frac{256 - \text{prob}}{256}$ — with $1 \leq \text{prob} \leq 255$ is as follows.

Using an unsigned 16-bit multiply followed by an unsigned right shift, we calculate an unsigned 8-bit split value:

$$split = 1 + (((\text{range} - 1) \cdot \text{probability}) \gg 8)$$

split is approximately $\left(\frac{\text{prob}}{256}\right) \cdot \text{range}$ and lies within the bounds $1 \leq \text{split} \leq \text{range} - 1$. These bounds ensure the correctness of the decoding procedure described below.

If val is false, we leave the left interval endpoint bottom alone and reduce range, replacing it by split. If val is true, we move up the left endpoint to bottom+split, propagating any carry to the already-written value w (this is where we need the ability to add 1 to w), and reduce range to range−split.

Regardless of the value encoded, range has been reduced and now has the bounds $1 \leq \text{range} \leq 254$. If $\text{range} < 128$, the encoder doubles it and shifts the high-order bit out of bottom to the output as it also doubles bottom, repeating this process one bit at a time until $128 \leq \text{range} \leq 255$. Once this is completed, the encoder is ready to accept another bool, maintaining the constraints described above.

After encoding the last bool, the partition may be completed by appending bottom to the bitstream.

The decoder mimics the state of the encoder. It maintains, together with an input bit position, two unsigned 8-bit numbers, a range identical to that maintained by the encoder and a value. Decoding one bool at a time, the decoder (in effect) tracks the same left interval endpoint as does the encoder and subtracts it from the remaining input. Appending the unread portion of the bitstream to the 8-bit value gives the difference between the actual value encoded and the known left endpoint.

The decoder is initialized by setting $\text{range} = 255$ and reading the first 16 input bits into value. The decoder maintains range and calculates split in exactly the same way as does the encoder.

To decode a bool, it compares value to split; if value<split, the bool is zero, and range is replaced with split. If value≥split, the bool is one, range is replaced with range−split, and value is replaced with value−split.

Again, range is doubled one bit at a time until it is at least 128. The value is doubled in parallel, shifting a new input bit into the bottom each time.

Writing Value for value together with the unread input bits and Range for range extended indefinitely on the right by zeros, the condition Value<Range is maintained at all times by the decoder. In particular, the bits shifted out of value as it is doubled are always zero.

7.3 Actual Implementation

The C code below gives complete implementations of the encoder and decoder described above. While they are logically identical to the “bit-at-a-time” versions, they internally buffer a couple of extra bytes of the bitstream. This allows I/O to be done (more practically) a byte at a time and drastically reduces the number of carries the encoder has to propagate into the already-written data.

Another (logically equivalent) implementation may be found in the reference decoder files `dboolhuff.h` and `dboolhuff.c`.

```
/* Encoder first */

typedef struct {
  uint8 *output;  /* ptr to next byte to be written */
  uint32 range;   /* 128 <= range <= 255 */
  uint32 bottom;  /* minimum value of remaining output */
  int bit_count;   /* # of shifts before an output byte is available */
} bool_encoder;

/* Must set initial state of encoder before writing any bools. */

void init_bool_encoder( bool_encoder *e, uint8 *start_partition)
{
  e->output = start_partition;
  e->range = 255;

  e->bottom = 0;
  e->bit_count = 24;
}

/* Encoding very rarely produces a carry that must be propagated to
   the already-written output. The arithmetic guarantees that the propagation
   will never go beyond the beginning of the output. Put another way, the
   encoded value x is always less than one. */

void add_one_to_output( uint8 *q)
{
  while( *--q == 255)
    *q = 0;
  ++*q;
}

/* Main function writes a bool_value whose probability of being zero is
   (expected to be) prob/256. */

void write_bool( bool_encoder *e, Prob prob, int bool_value)
{
```

```
  /* split is approximately (range * prob) / 256  and, crucially,
     is strictly bigger than zero and strictly smaller than range */

  uint32 split = 1 + ( ((e->range - 1) * prob) >> 8);

  if( bool_value) {
    e->bottom += split; /* move up bottom of interval */
    e->range -= split;  /* with corresponding decrease in range */
  } else
    e->range = split;   /* decrease range, leaving bottom alone */

  while( e->range < 128)
  {
    e->range <<= 1;

    if( e->bottom & (1 << 31))  /* detect carry */
      add_one_to_output( e->output);

    e->bottom <<= 1;            /* before shifting bottom */

    if( !--e->bit_count) {       /* write out high byte of bottom ... */

      *e->output++ = (uint8) (e->bottom >> 24);

      e->bottom &= (1 << 24) - 1;     /* ... keeping low 3 bytes */

      e->bit_count = 8;               /* 8 shifts until next output */
    }
  }
}

/* Call this function (exactly once) after encoding the last bool value
   for the partition being written */

void flush_bool_encoder( bool_encoder *e)
{
  int c = e->bit_count;
  uint32 v = e->bottom;

  if( v & (1 << (32 - c)))        /* propagate (unlikely) carry */
    add_one_to_output( e->output);
  v <<= c & 7;                    /* before shifting remaining output */
  c >>= 3;                        /* to top of internal buffer */
  while( --c >= 0)
    v <<= 8;
  c = 4;
  while( --c >= 0) {            /* write remaining data, possibly padded */
    *e->output++ = (uint8) (v >> 24);
    v <<= 8;
  }
}

/* Decoder state exactly parallels that of the encoder.
   "value", together with the remaining input, equals the complete encoded
   number x less the left endpoint of the current coding interval. */
```

```
typedef struct {
  uint8   *input;      /* pointer to next compressed data byte */
  uint32  range;       /* always identical to encoder's range */
  uint32  value;       /* contains at least 24 significant bits */
  int     bit_count;   /* # of bits shifted out of value, at most 7 */
} bool_decoder;

/* Call this function before reading any bools from the partition.*/

void init_bool_decoder( bool_decoder *d, uint8 *start_partition)
{
  {
    int i = 0;
    d->value = 0;           /* value = first 24 input bytes */
    while( ++i <= 24)
      d->value = (d->value << 8)  |  *start_partition++;
  }

  d->input = start_partition;  /* ptr to next byte to be read */
  d->range = 255;              /* initial range is full */
  d->bit_count = 0;            /* have not yet shifted out any bits */
}

/* Main function reads a bool encoded at probability prob/256, which of
   course must agree with the probability used when the bool was written. */

int read_bool( bool_decoder *d, Prob prob)
{
  /* range and split are identical to the corresponding values used
  by the encoder when this bool was written */

  uint32  split = 1 + ( ((d->range - 1) * prob) >> 8);
  uint32  SPLIT = split << 8;
  int     retval;              /* will be 0 or 1 */

  if( d->value >= SPLIT) {     /* encoded a one */
    retval = 1;
    d->range -= split;         /* reduce range */
    d->value -= SPLIT;         /* subtract off left endpoint of interval */
  } else {                     /* encoded a zero */
    retval = 0;
    d->range = split;          /* reduce range, no change in left endpoint */
  }

  while( d->range < 128) {     /* shift out irrelevant value bits */
    d->value <<= 1;
    d->range <<= 1;
    if( ++d->bit_count == 8) { /* shift in new bits 8 at a time */
      d->bit_count = 0;
      d->value |= *d->input++;
    }
  }
  return retval;
}

/* Convenience function reads a "literal", that is, a "num_bits" wide
```

```
   unsigned value whose bits come high- to low-order, with each bit
   encoded at probability 128 (i.e., 1/2). */

uint32 read_literal( bool_decoder *d, int num_bits)
{
  uint32 v = 0;
  while( num_bits--)
    v = (v << 1) + read_bool( d, 128);
  return v;
}

/* Variant reads a signed number */

int32 read_signed_literal( bool_decoder *d, int num_bits)
{
  int32 v = 0;
  if( !num_bits)
    return 0;
  if( read_bool( d, 128))
    v = -1;
  while( --num_bits)
    v = (v << 1) + read_bool( d, 128);
  return v;
}
```

Chapter 8: Compressed Data Components

At the lowest level, VP8's compressed data is simply a sequence of probabilistically-encoded bools. Most of this data is composed of (slightly) larger semantic units fashioned from bools, which we describe here.

We sometimes use these descriptions in C expressions within data format specifications. In this context, they refer to the return value of a call to an appropriate bool_decoder d, reading (as always) from its current reference point.

Call	Alt.	Return
`Bool(p)`	`B(p)`	Bool with probability `p` of being `0`. Abbreviated `B(p)`. Return value of `read_bool(d, p)`.
`Flag`	`F`	A one-bit flag (same thing as a `B(128)` or an `L(1)`). Abbreviated `F`. Return value of `read_bool(d, 128)`.
`Lit(n)`	`L(n)`	Unsigned n-bit number encoded as n flags (a "literal"). Abbreviated `L(n)`. The bits are read from high to low order. Return value of `read_literal(d, n)`.
`SignedLit(n)`		Signed n-bit number encoded similarly to an `L(n)`. Return value of `read_signed_literal(d, n)`. These are rare.
`P(8)`		An 8-bit probability. No different from an `L(8)`, but we sometimes use this notation to emphasize that a probability is being coded.
`P(7)`		A 7-bit specification of an 8-bit probability. Coded as an `L(7)` number `x`; the resulting 8-bit probability is $x ? x << 1 : 1$.
`F? X`		A flag which, if true, is followed by a piece of data `X`.
`F? X:Y`		A flag which, if true, is followed by `X` and, if false, is followed by `Y`. Also used to express a value where `Y` is an implicit default (not encoded in the data stream), as in $F ? P(8) : 255$, which expresses an optional probability: if the flag is true, the probability is specified as an 8-bit literal, while if the flag is false, the probability defaults to `255`.
`B(p)? X`	`B(p)? X:Y`	Variants of the above using a boolean indicator whose probability is not necessarily `128`.
`X`		Multi-component field, the specifics of which will be given at a more appropriate point in the discussion.
`T`		Tree-encoded value from small alphabet.

The last type requires elaboration. We often wish to encode something whose value is restricted to a small number of possibilities (the alphabet).

This is done by representing the alphabet as the leaves of a small binary tree. The (non-leaf) nodes of the tree have associated probabilities `p` and correspond to calls to `read_bool(d, p)`. We think of a zero as choosing the left branch below the node and a one as choosing the right branch.

Thus every value (leaf) whose tree depth is `x` is decoded after exactly `x` calls to `read_bool`.

A tree representing an encoding of an alphabet of `n` possible values always contains `n-1` non-leaf nodes, regardless of its shape (this is easily seen by induction on `n`).

There are many ways that a given alphabet can be so represented. The choice of tree has little impact on datarate but does affect decoder performance. The trees used by VP8 are chosen to (on average) minimize the number of calls to `read_bool`. This amounts to shaping the tree so that more probable values have smaller tree depth than do less probable values.

Readers familiar with Huffman coding will notice that, given an alphabet together with probabilities for each value, the associated Huffman tree minimizes the expected number of calls to `read_bool`. Such readers will also realize that the coding method described here never results in higher datarates than does the Huffman method and, indeed, often results in much lower datarates. Huffman coding is, in fact, nothing more than a special case of this method in which each node probability is fixed at 128 (i.e., 1/2).

8.1 Tree Coding Implementation

We give a suggested implementation of a tree data structure followed by a couple of actual examples of its usage by VP8.

It is most convenient to represent the values using small positive integers, typically an enum counting up from zero. The largest alphabet (used to code DCT coefficients, described in Chapter 13 that is tree-coded by VP8 has only 12 values. The tree for this alphabet adds 11 interior nodes and so has a total of 23 positions. Thus, an 8-bit number easily accommodates both a tree position and a return value.

A tree may then be compactly represented as an array of (pairs of) 8-bit integers. Each (even) array index corresponds to an interior node of the tree;, the zeroth index of course corresponds to the root of the tree. The array entries come in pairs corresponding to the left (0) and right (1) branches of the subtree below the interior node. We use the convention that a positive (even) branch entry is the index of a deeper interior node, while a nonpositive entry v corresponds to a leaf whose value is $-v$.

The node probabilities associated to a tree-coded value are stored in an array whose indices are half the indices of the corresponding tree positions. The length of the probability array is one less than the size of the alphabet.

Here is C code implementing the foregoing. The advantages of our data structure should be noted. Aside from the smallness of the structure itself, the tree-directed reading algorithm is essentially a single line of code.

```
/* A tree specification is simply an array of 8-bit integers. */

typedef int8 tree_index;
typedef const tree_index Tree[];

/* Read and return a tree-coded value at the current decoder position. */

int treed_read(
  bool_decoder * const d,      /* bool_decoder always returns a 0 or 1 */
  Tree t,                     /* tree specification */
```

```
  const Prob p[]              /* corresponding interior node probabilities */
) {
  register tree_index i = 0;   /* begin at root */

  /* Descend tree until leaf is reached */

  while( ( i = t[ i + read_bool( d, p[i>>1]) ] ) > 0) {}

  return -i;      /* return value is negation of nonpositive index */
}
```

Tree-based decoding is implemented in the reference decoder file `tree_reader.h`.

8.2 Tree Coding Example

As a multi-part example, without getting too far into the semantics of macroblock decoding (which is of course taken up below), we look at the “mode” coding for intra-predicted macroblocks.

It so happens that, because of a difference in statistics, the Y (or luma) mode encoding uses two different trees: one for key frames and another for interframes. This is the only instance in VP8 of the same dataset being coded by different trees under different circumstances. The UV (or chroma) modes are a proper subset of the Y modes and, as such, have their own decoding tree.

```
typedef enum
{
    DC_PRED,    /* predict DC using row above and column to the left */
    V_PRED,     /* predict rows using row above */
    H_PRED,     /* predict columns using column to the left */
    TM_PRED,    /* propagate second differences a la "true motion" */

    B_PRED,     /* each Y subblock is independently predicted */

    num_uv_modes = B_PRED,  /* first four modes apply to chroma */
    num_ymodes   /* all modes apply to luma */
}
intra_mbmode;

/* The aforementioned trees together with the implied codings as comments.
   Actual (i.e., positive) indices are always even.
   Value (i.e., nonpositive) indices are arbitrary. */

const tree_index ymode_tree [2 * (num_ymodes - 1)] =
{
 -DC_PRED, 2,           /* root: DC_PRED = "0", "1" subtree */
  4, 6,                 /* "1" subtree has 2 descendant subtrees */
   -V_PRED, -H_PRED,    /* "10" subtree:  V_PRED = "100", H_PRED = "101" */
   -TM_PRED, -B_PRED    /* "11" subtree:  TM_PRED = "110", B_PRED = "111" */
};

const tree_index kf_ymode_tree [2 * (num_ymodes - 1)] =
{
 -B_PRED, 2,            /* root: B_PRED = "0", "1" subtree */
```

```
  4, 6,                 /* "1" subtree has 2 descendant subtrees */
   -DC_PRED, -V_PRED,   /* "10" subtree: DC_PRED = "100", V_PRED = "101" */
   -H_PRED, -TM_PRED    /* "11" subtree: H_PRED = "110", TM_PRED = "111" */
};

const tree_index uv_mode_tree [2 * (num_uv_modes - 1)] =
{
 -DC_PRED, 2,           /* root: DC_PRED = "0", "1" subtree */
  -V_PRED, 4,           /* "1" subtree:  V_PRED = "10", "11" subtree */
   -H_PRED, -TM_PRED    /* "11" subtree: H_PRED = "110", TM_PRED = "111" */
};

/* Given a bool_decoder d, a Y mode might be decoded as follows.*/

const Prob pretend_its_huffman [num_ymodes - 1] = { 128, 128, 128, 128};
Ymode = (intra_mbmode) treed_read( d, ymode_tree, pretend_its_huffman);
```

Since it greatly facilitates re-use of reference code and since there is no real reason to do otherwise, it is strongly suggested that any decoder implementation use exactly the same enumeration values and probability table layouts as described in this document (and in the reference code) for all tree-coded data in VP8.

Chapter 9: Frame Header

The uncompressed data chunk at the start of each frame and the first part of the first data partition contains information pertaining to the frame as a whole. We list the fields in the order of occurrence, giving details for some of the fields. Other details are postponed until a more logical point in our overall description. Most of the header decoding occurs in the reference decoder file `decodeframe.c`.

9.1 Uncompressed Data Chunk

The uncompressed data chunk comprises a common (for key frames and interframes) 3-byte frame tag that contains four fields, as follows:

1. A 1-bit frame type (`0` for key frames, `1` for interframes).
2. A 3-bit version number (`0` - `3` are defined as four different profiles with different decoding complexity; other values may be defined for future variants of the VP8 data format).
3. A 1-bit `show_frame` flag (`0` when current frame is not for display, `1` when current frame is for display).
4. A 19-bit field containing the size of the first data partition in bytes.

For key frames this is followed by a further 7 bytes of uncompressed data as follows:

```
Start code byte 0     0x9d
Start code byte 1     0x01
Start code byte 2     0x2a

16 bits       :       (2 bits Horizontal Scale << 14) | Width (14 bits)
16 bits       :       (2 bits Vertical Scale << 14) | Height (14 bits)
```

The following source code segment illustrates validation of the start code and reading the width, height and scale factors for a key frame.

```
unsigned char *c = pbi->Source+3;

// vet via sync code
if(c[0]!=0x9d||c[1]!=0x01||c[2]!=0x2a)
    return -1;
```

where `pbi->source` points to the beginning of the frame.

The following code reads the image dimension from the bitstream:

```
pc->Width      = swap2(*(unsigned short*)(c+3))&0x3fff;
pc->horiz_scale = swap2(*(unsigned short*)(c+3))>>14;
pc->Height     = swap2(*(unsigned short*)(c+5))&0x3fff;
pc->vert_scale  = swap2(*(unsigned short*)(c+5))>>14;
```

where `swap2` macro takes care of the endian on different platform:

```
#if defined(__ppc__) || defined(__ppc64__)
# define swap2(d)  \
  ((d&0x000000ff)<<8) |  \
  ((d&0x0000ff00)>>8)
#else
  # define swap2(d) d
#endif
```

While each frame is encoded as a raster scan of 16x16 macroblocks, the frame dimensions are not necessarily evenly divisible by 16. In this case, write `ew = 16 - (width & 15)` and `eh = 16 - (height & 15)` for the excess width and height, respectively. Although they are encoded, the last `ew` columns and `eh` rows are not actually part of the image and should be discarded before final output. However, these “excess pixels” should be maintained in the internal reconstruction buffer used to predict ensuing frames.

The scaling specifications for each dimension are encoded as follows.

Value	Scaling
`0`	No upscaling (the most common case).
`1`	Upscale by 5/4.
`2`	Upscale by 5/3.
`3`	Upscale by 2.

Upscaling does **not** affect the reconstruction buffer, which should be maintained at the encoded resolution. Any reasonable method of upsampling (including any that may be supported by video hardware in the playback environment) may be used. Since scaling has no effect on decoding, we do not discuss it any further.

As discussed in Chapter 5, allocation (or re-allocation) of data structures (such as the reconstruction buffer) whose size depends on dimension will be triggered here.

9.2 Color Space and Pixel Type (Key Frames-only)

Field	Value
`L(1)`	1-bit color space type specification
`L(1)`	1-bit pixel value clamping specification

The color space type bit is encoded as the following:

- `0` - YUV color space similar to the YCrCb color space defined in ITU-R BT.601
- `1` - YUV color space whose digital conversion to RGB does not involve multiplication and division

It should be noted that in either case, the actual conversion between YUV and RGB is not part of this specification.

Note: In the initial release of VP8 only color space type `0` is supported.

The pixel value clamping type bit is encoded as the following:

- `0` - Decoders are required to clamp the reconstructed pixel values to between `0` and `255` (inclusive).
- `1` - Reconstructed pixel values are guaranteed to be between `0` and `255`, no clamping is necessary.

Information in this subsection does not appear in interframes.

9.3 Segment-based Adjustments

This subsection contains probability and value information for implementing segment adaptive adjustments to default decoder behaviors. The data in this section is used in the decoding of the ensuing per-segment information and applies to the entire frame. When segment adaptive adjustments are enabled, each macroblock will be assigned a segment ID. Macroblocks with the same segment ID belong to same segment, and have the same adaptive adjustments over default baseline values for the frame. The adjustments can be quantization level or loop filter strength.

The context for decoding this feature is provided by section B of the frame header. It contains:

1. A `segmentation_enabled` Flag which if `1` (`0`), enables (disables) the feature for this frame. The remaining fields occur if the feature is enabled.
2. `L(1)` indicates if the segment map is updated for the current frame (`update_mb_segmentaton_map`)
3. `L(1)` indicates if the segment feature data items are updated for the current frame
4. If flag in 3 is `1`, the following fields occur:
    1. `L(1)` the mode of segment feature data, can be absolute value mode or delta value mode, later mode, feature data is the difference against current frame defaults.
    2. Segment feature data items are decoded segment by each segment for each segment feature. For every data item, a one bit flag indicating if the item is `0` or a non-zero value to be decoded. If there is non-zero value, the value is decoded as a magnitude `L(n)` followed by a one bit sign (`L(1)`, `0` for positive and `1` for negative). The length `n` can be looked up from a pre-defined length table for all feature data.
5. If flag in 2 is `1`, the probabilities of the decoding tree for segment map are decoded from the bitstream. Each probability is decoded with one bit flag indicating if the probability is the default value of `255` (flag is `0`), or the probability is an 8-bit value, `L(8)`, from the bitstream.

The layout and semantics supporting this feature at the macroblock level will be described in Chapter 10.

9.4 Loop Filter Type and Levels

VP8 supports two types of loop filter, having different computational complexity. The following bits occur in the header to support the selection of the baseline type, strength and sharpness behavior of the loop filter used for the current frame.

Index	Description
`L(1)`	filter_type
`L(6)`	loop_filter_level
`L(3)`	sharpness_level

The meaning of these numbers will be further explained in Chapter 15.

VP8 has a feature in the bitstream that enables adjustment of the loop filter level based on a macroblock's prediction mode and reference frame. The per-macroblock adjustment is done through delta values against default loop filter level for the current frame. This subsection contains flag and value information for implementing per-macroblock loop filter level adjustment to default decoder behaviors. The data in this section is used in the decoding of the ensuing per-macroblock information and applies to the entire frame.

`L(1)` is a one-bit flag indicating if macroblock loop filter adjustment is on for the current frame. `0` means such feature is not supported in the current frame and `1` means this feature is enabled for the current frame.

Whether the adjustment is based on reference frame or encoding mode, the adjustment of loop filter level is done via a delta value against a baseline loop filter value. The delta values are updated for the current frame if an `L(1)` bit, `mode_ref_lf_delta_update`, takes the value `1`. There are two groups of delta values, one group of delta values are for reference frame-based adjustments, the other group is for mode-based adjustments. The number of delta values in the two groups is `MAX_REF_LF_DELTAS` and `MAX_MODE_LF_DELTAS`, respectively. For every value within the two groups, there is one bit `L(1)` to indicate if the particular value is updated. When one is updated (1), it is transmitted as a six-bit magnitude `L(6)` followed by a one-bit sign flag `(L(1)`, `0` for positive and `1` for negative).

9.5 Token Partition and Partition Data Offsets

VP8 allows DCT coefficients to be packed into multiple partitions besides the first partition with header and per-macroblock prediction information, so the decoder can perform parallel decoding in an efficient manner. There are two bits `L(2)` used to indicate the number of coefficient data partitions within a compressed frame. The two bits are defined in the following table:

Bit 1	Bit 0	Number of Partitions
0	0	1
0	1	2
1	0	4
1	1	8

When the number of partitions is greater than one, offsets are embedded in the bitstream to provide the decoder direct access to token partitions. Each offset is written in 3 bytes (24 bits). Since the offset to the first partition is always 0, only the offsets for partitions other than the first partition are encoded in the bitstream. The partitioned data are consecutive in the bitstream, so offsets can also be used to calculate the data size of each partition. The following pseudo code illustrates how the size/offset is defined by the three bytes in the bitstream.

```
Offset/size  =  (uint32)(byte0) + ((uint32)(byte1)<<8) + ((uint32)(byte2)<<16);
```

9.6 Dequantization Indices

All residue signals are specified via a quantized 4x4 DCT applied to the Y, U, V, or Y2 subblocks of a macroblock. As detailed in Chapter 14, before inverting the transform, each decoded coefficient is multiplied by one of six dequantization factors, the choice of which depends on the plane (Y, chroma = U or V, Y2) and coefficient position (DC = coefficient 0, AC = coefficients 1-15). The six values are specified using 7-bit indices into six corresponding fixed tables (the tables are given in Chapter 14).

The first 7-bit index gives the dequantization table index for Y plane AC coefficients, called yac_qi. It is always coded and acts as a baseline for the other 5 quantization indices, each of which is represented by a delta from this baseline index. Following is pseudo code for reading the indices:

```
yac_qi     = L(7);           /* Y ac index always specified */
ydc_delta  = F? delta(): 0;  /* Y dc delta specified if flag is true */

y2dc_delta = F? delta(): 0;  /* Y2 dc delta specified if flag is true */
y2ac_delta = F? delta(): 0;  /* Y2 ac delta specified if flag is true */

uvdc_delta = F? delta(): 0;  /* chroma dc delta specified if flag is true */
uvac_delta = F? delta(): 0;  /* chroma ac delta specified if flag is true */
```

Where `delta()` is the process to read 5 bits from the bitstream to determine a signed delta value:

Index	Description
`L(4)`	Magnitude of delta
`L(1)`	Sign of delta, 0 for positive and 1 for negative

9.7 Refresh Golden Frame and AltRef Frame

For key frames, both golden frame and altref frame are refreshed/replaced by the current reconstructed frame, by default. For non-key frames, VP8 uses two bits to indicate whether the two frame buffers are refreshed, using the reconstructed current frame:

Index	Description
`L(1)`	Whether golden frame is refreshed (`0` complete. This flag does not occur for no, `1` for yes).
`L(1)`	Whether altref frame is refreshed (`0` for no, `1` for yes).

When the flag for golden frame is `0`, VP8 uses 2 more bits in the bitstream to indicate whether the buffer (and which buffer) is copied to the golden frame, or if no buffer is copied:

Index	Description
`L(2)`	Buffer copy flag for golden frame buffer

Where:

- `0` means no buffer is copied to golden frame
- `1` means last_frame is copied to golden frame
- `2` means alt_ref_frame is copied to golden frame

Similarly, when the flag for altref is `0`, VP8 uses 2 bits in the bitstream to indicate which buffer is copied to `alt_ref_frame`.

Index	Description
`L(2)`	Buffer copy flag for altref frame buffer

Where:

- `0` means no buffer is copied to altref frame
- `1` means `last_frame` is copied to altref frame
- `2` means `golden_frame` is copied to altref frame

Two bits are transmitted for `ref_frame_sign_bias` for `golden_frame` and `alt_ref_frame` respectively.

Index	Description
`L(1)`	Sign bias flag for golden frame
`L(1)`	Sign bias flag for altref frame

These values are used to control the sign of the motion vectors when a golden frame or an altref frame is used as the reference frame for a macroblock.

9.8 Refresh Last Frame Buffer

VP8 uses one bit, `L(1)`, to indicate if the last frame reference buffer is refreshed using the constructed current frame. On key frame this bit is overridden, and the last frame buffer is always refreshed.

9.9 DCT Coefficient Probability Update

Contains a partial update of the probability tables used to decode DCT coefficients. These tables are maintained across interframes but are of course replaced with their defaults at the beginning of every key frame.

The layout and semantics of this field will be taken up in Chapter 13.

9.10 Remaining Frame Header Data (non-Key Frame)

Index	Description
`L(1)`	`mb_no_coeff_skip`. This flag indicates at the frame level if skipping of macroblocks with no non-zero coefficients is enabled. If it is set to `0` then `prob_skip_false` is not read and `mb_skip_coeff` is forced to `0` for all macroblocks (see Sections 11.1 and 12.1).
`L(8)`	`prob_skip_false` = probability used for decoding a macroblock level flag, which indicates if a macroblock has any non-zero coefficients. Only read if `mb_no_coeff_skip` is `1`.
`L(8)`	`prob_intra` = probability that a macroblock is “intra” predicted, that is, predicted from the already-encoded portions of the current frame as opposed to “inter” predicted, that is, predicted from the contents of a prior frame.
`L(8)`	`prob_last` = probability that an inter-predicted macroblock is predicted from the immediately previous frame, as opposed to the most recent golden frame or altref frame..
`L(8)`	`prob_gf` = probability that an inter-predicted macroblock is predicted from the most recent golden frame, as opposed to the altref frame
`F`	If true, followed by four `L(8)` s updating the probabilities for the different types of intra-prediction for the Y plane. These probabilities correspond to the four interior nodes of the decoding tree for intra Y modes in an interframe, that is, the even positions in the `ymode_tree` array given above.
`F`	If true, followed by three `L(8)` s updating the probabilities for the different types of intra-prediction for the chroma planes. These probabilities correspond to the even positions in the `uv_mode_tree` array given above.
`X`	Motion vector probability update. The details will be given after the discussion of motion vector decoding.

Decoding of this portion (only) of the frame header is handled in the reference decoder file `decodemv.c`.

9.11 Remaining Frame Header Data (Key Frame)

Index	Description
`L(1)`	`mb_no_coeff_skip`. This flag indicates at the frame level if skipping of macroblocks with no non-zero coefficients is enabled. If it is set to `0` then `prob_skip_false` is not read and `mb_skip_coeff` is forced to `0` for all macroblocks (see Sections 11.1 and 12.1).
`L(8)`	`prob_skip_false` = Probability used for decoding a macroblock level flag, which indicates if a macroblock has any non-zero coefficients. Only read if `mb_no_coeff_skip` is `1`.

Decoding of this portion of the frame header is handled in the reference decoder file `demode.c`.

This completes the layout of the frame header. The remainder of the first data partition consists of macroblock-level prediction data.

After the frame header is processed, all probabilities needed to decode the prediction and residue data are known and will not change until the next frame.

Chapter 10: Segment-based Feature Adjustments

Every macroblock may optionally override some of the default behaviors of the decoder. Specifically, VP8 uses segment based adjustments to support changing quantizer level and loop filter level for a macroblock. When the segment-based adjustment feature is enabled for a frame, each macroblock within the frame is coded with a `segment_id`. This effectively segments all the macroblocks in the current frame into a number of different segments. Macroblocks within the same segment behave exactly the same for quantizer and loop filter level adjustments.

If both the `segmentation_enabled` and `update_mb_segmentation_map` flags in subsection B of the frame header take a value of `1`, the prediction data for each (intra- or inter-coded) macroblock begins with a specification of `segment_id` for the current macroblock. It is decoded using this simple tree …

```
const tree_index mb_segment_tree [2 * (4-1)] =
  {
    2,  4,      /* root: "0", "1" subtrees */
    -0, -1,     /* "00" = 0th value, "01" = 1st value */
     -2, -3     /* "10" = 2nd value, "11" = 3rd value */
  }
```

… combined with a 3-entry probability table `mb_segment_tree_probs[3]`. The macroblock's `segment_id` is used later in the decoding process to look into the `segment_feature_data` table and determine how the quantizer and loop filter levels are adjusted.

The decoding of `segment_id`, together with the parsing of intra-prediction modes (which is taken up next), is implemented in the reference decoder file `demode.c`.

Chapter 11: Key Frame Macroblock Prediction Records

After the features described above, the macroblock prediction record next specifies the prediction mode used for the macroblock.

11.1 mb_skip_coeff

The single bool flag is decoded using `prob_skip_false` if and only if `mb_no_coeff_skip` is set to `1` (see sections 9.10 and 9.11). If `mb_no_coeff_skip` is set to `0` then this value defaults to `0`.

11.2 Luma Modes

First comes the luma specification of type `intra_mbmode`, coded using the `kf_ymode_tree`, as described in Chapter 8 and repeated here for convenience:

```
typedef enum
{
    DC_PRED,    /* predict DC using row above and column to the left */
    V_PRED,     /* predict rows using row above */
    H_PRED,     /* predict columns using column to the left */
    TM_PRED,    /* propagate second differences a la "true motion" */

    B_PRED,     /* each Y subblock is independently predicted */

    num_uv_modes = B_PRED,  /* first four modes apply to chroma */
    num_ymodes   /* all modes apply to luma */
}
intra_mbmode;

const tree_index kf_ymode_tree [2 * (num_ymodes - 1)] =
{
 -B_PRED, 2,            /* root: B_PRED = "0", "1" subtree */
  4, 6,                 /* "1" subtree has 2 descendant subtrees */
   -DC_PRED, -V_PRED,   /* "10" subtree: DC_PRED = "100", V_PRED = "101" */
   -H_PRED, -TM_PRED    /* "11" subtree: H_PRED = "110", TM_PRED = "111" */
};
```

For key frames, the Y mode is decoded using a fixed probability array as follows:

```
const Prob kf_ymode_prob [num_ymodes - 1] = { 145, 156, 163, 128};
Ymode = (intra_mbmode) treed_read( d, kf_ymode_tree, kf_ymode_prob);
```

`d` is of course the `bool_decoder` being used to read the first data partition.

If the Ymode is `B_PRED`, it is followed by a (tree-coded) mode for each of the 16 Y subblocks. The 10 subblock modes and their coding tree as follows:

```
typedef enum
{
```

```
    B_DC_PRED,  /* predict DC using row above and column to the left */
    B_TM_PRED,  /* propagate second differences a la "true motion" */

    B_VE_PRED,  /* predict rows using row above */
    B_HE_PRED,  /* predict columns using column to the left */

    B_LD_PRED,  /* southwest (left and down) 45 degree diagonal prediction */
    B_RD_PRED,  /* southeast (right and down) "" */

    B_VR_PRED,  /* SSE (vertical right) diagonal prediction */
    B_VL_PRED,  /* SSW (vertical left) "" */
    B_HD_PRED,  /* ESE (horizontal down) "" */
    B_HU_PRED,  /* ENE (horizontal up) "" */

    num_intra_bmodes
}
intra_bmode;

/* Coding tree for the above, with implied codings as comments */

const tree_index bmode_tree [2 * (num_intra_bmodes - 1)] =
{
 -B_DC_PRED, 2,                 /* B_DC_PRED = "0" */
  -B_TM_PRED, 4,                /* B_TM_PRED = "10" */
   -B_VE_PRED, 6,               /* B_VE_PRED = "110" */
    8, 12,
     -B_HE_PRED, 10,            /* B_HE_PRED = "1110" */
      -B_RD_PRED, -B_VR_PRED,   /* B_RD_PRED = "111100", B_VR_PRED = "111101" */
     -B_LD_PRED, 14,            /* B_LD_PRED = "111110" */
       -B_VL_PRED, 16           /* B_VL_PRED = "1111110" */
         -B_HD_PRED, -B_HU_PRED /* HD = "11111110", HU = "11111111" */
};
```

The first four modes are smaller versions of the similarly-named 16x16 modes above, albeit with slightly different numbering. The last six “diagonal” modes are unique to luma subblocks.

11.3 Subblock Mode Contexts

The coding of subblock modes in key frames uses the modes already coded for the subblocks to the left of and above the subblock to select a probability array for decoding the current subblock mode. This is our first instance of contextual prediction, and there are several caveats associated with it:

1. The adjacency relationships between subblocks are based on the normal default raster placement of the subblocks.

2. The adjacent subblocks need not lie in the current macroblock. The subblocks to the left of the left-edge subblocks 0, 4, 8, and 12 are the right-edge subblocks 3, 7, 11, and 15, respectively, of the (already coded) macroblock immediately to the left. Similarly, the subblocks above the top-edge subblocks 0, 1, 2, and 3 are the bottom-edge subblocks 12, 13, 14, and 15 of the already-coded macroblock immediately above us.

3. For macroblocks on the top row or left edge of the image, some of the predictors will be non-existent. Such predictors are taken to have had the value `B_DC_PRED` which, perhaps conveniently, takes the

value `0` in the enumeration above. A simple management scheme for these contexts might maintain a row of above predictors and four left predictors. Before decoding the frame, the entire row is initialized to `B_DC_PRED`; before decoding each row of macroblocks, the four left predictors are also set to `B_DC_PRED`. After decoding a macroblock, the bottom four subblock modes are copied into the row predictor (at the current position, which then advances to be above the next macroblock) and the right four subblock modes are copied into the left predictor.

4. Many macroblocks will of course be coded using a 16x16 luma prediction mode. For the purpose of predicting ensuing subblock modes (only), such macroblocks derive a subblock mode, constant throughout the macroblock, from the 16x16 luma mode as follows: `DC_PRED` uses `B_DC_PRED`, `V_PRED` uses `B_VE_PRED`, `H_PRED` uses `B_HE_PRED`, and `TM_PRED` uses `B_TM_PRED`.

5. Although we discuss interframe modes later, we remark here that, while interframes do use all the intra-coding modes described here and below, the subblock modes in an interframe are coded using a single constant probability array that does not depend on any context.

The dependence of subblock mode probability on the nearby subblock mode context is most easily handled using a three-dimensional constant array:

```
const Prob kf_bmode_prob [num_intra_bmodes] [num_intra_bmodes] [num_intra_bmodes-1];
```

The outer two dimensions of this array are indexed by the already-coded subblock modes above and to the left of the current block, respectively. The inner dimension is a typical tree probability list whose indices correspond to the even indices of the `bmode_tree` above. The mode for the j^{th} luma subblock is then

```
Bmode = (intra_bmode) treed_read( d, bmode_tree, kf_bmode_prob [A] [L]);
```

where the 4x4 Y subblock index j varies from `0` to `15` in raster order and `A` and `L` are the modes used above and to-the-left of the j^{th} subblock.

The contents of the `kf_bmode_prob` array are given at the end of this chapter.

11.4 Chroma Modes

After the Y mode (and optional subblock mode) specification comes the chroma mode. The chroma modes are a subset of the Y modes and are coded using the `uv_mode_tree` described in Chapter 8, again repeated here for convenience:

```
const tree_index uv_mode_tree [2 * (num_uv_modes - 1)] =
{
 -DC_PRED, 2,            /* root: DC_PRED = "0", "1" subtree */
  -V_PRED, 4,            /* "1" subtree:  V_PRED = "10", "11" subtree */
   -H_PRED, -TM_PRED     /* "11" subtree: H_PRED = "110", TM_PRED = "111" */
};
```

As for the Y modes (in a key frame), the chroma modes are coded using a fixed, contextless probability table:

```
const Prob kf_uv_mode_prob [num_uv_modes - 1] = { 142, 114, 183};
uv_mode = (intra_mbmode) treed_read( d, uv_mode_tree, kf_uv_mode_prob);
```

This completes the description of macroblock prediction coding for key frames. As will be discussed in Chapter 16, the coding of intra modes within interframes is similar, but not identical, to that described here (and in the reference code) for prediction modes and, indeed, for all tree-coded data in VP8.

11.5 Subblock Mode Probability Table

Finally, here is the fixed probability table used to decode subblock modes in key frames.

```
const Prob kf_bmode_prob [num_intra_bmodes] [num_intra_bmodes] [num_intra_bmodes-1] =
{
  {
    { 231, 120,  48,  89, 115, 113, 120, 152, 112},
    { 152, 179,  64, 126, 170, 118,  46,  70,  95},
    { 175,  69, 143,  80,  85,  82,  72, 155, 103},
    {  56,  58,  10, 171, 218, 189,  17,  13, 152},
    { 144,  71,  10,  38, 171, 213, 144,  34,  26},
    { 114,  26,  17, 163,  44, 195,  21,  10, 173},
    { 121,  24,  80, 195,  26,  62,  44,  64,  85},
    { 170,  46,  55,  19, 136, 160,  33, 206,  71},
    {  63,  20,   8, 114, 114, 208,  12,   9, 226},
    {  81,  40,  11,  96, 182,  84,  29,  16,  36}
  },
  {
    { 134, 183,  89, 137,  98, 101, 106, 165, 148},
    {  72, 187, 100, 130, 157, 111,  32,  75,  80},
    {  66, 102, 167,  99,  74,  62,  40, 234, 128},
    {  41,  53,   9, 178, 241, 141,  26,   8, 107},
    { 104,  79,  12,  27, 217, 255,  87,  17,   7},
    {  74,  43,  26, 146,  73, 166,  49,  23, 157},
    {  65,  38, 105, 160,  51,  52,  31, 115, 128},
    {  87,  68,  71,  44, 114,  51,  15, 186,  23},
    {  47,  41,  14, 110, 182, 183,  21,  17, 194},
    {  66,  45,  25, 102, 197, 189,  23,  18,  22}
  },
  {
    {  88,  88, 147, 150,  42,  46,  45, 196, 205},
    {  43,  97, 183, 117,  85,  38,  35, 179,  61},
    {  39,  53, 200,  87,  26,  21,  43, 232, 171},
    {  56,  34,  51, 104, 114, 102,  29,  93,  77},
    { 107,  54,  32,  26,  51,   1,  81,  43,  31},
    {  39,  28,  85, 171,  58, 165,  90,  98,  64},
    {  34,  22, 116, 206,  23,  34,  43, 166,  73},
    {  68,  25, 106,  22,  64, 171,  36, 225, 114},
    {  34,  19,  21, 102, 132, 188,  16,  76, 124},
    {  62,  18,  78,  95,  85,  57,  50,  48,  51}
  },
  {
    { 193, 101,  35, 159, 215, 111,  89,  46, 111},
    {  60, 148,  31, 172, 219, 228,  21,  18, 111},
```

```
    { 112, 113,  77,  85, 179, 255,  38, 120, 114},
    {  40,  42,   1, 196, 245, 209,  10,  25, 109},
    { 100,  80,   8,  43, 154,   1,  51,  26,  71},
    {  88,  43,  29, 140, 166, 213,  37,  43, 154},
    {  61,  63,  30, 155,  67,  45,  68,   1, 209},
    { 142,  78,  78,  16, 255, 128,  34, 197, 171},
    {  41,  40,   5, 102, 211, 183,   4,   1, 221},
    {  51,  50,  17, 168, 209, 192,  23,  25,  82}
  },
  {
    { 125,  98,  42,  88, 104,  85, 117, 175,  82},
    {  95,  84,  53,  89, 128, 100, 113, 101,  45},
    {  75,  79, 123,  47,  51, 128,  81, 171,   1},
    {  57,  17,   5,  71, 102,  57,  53,  41,  49},
    { 115,  21,   2,  10, 102, 255, 166,  23,   6},
    {  38,  33,  13, 121,  57,  73,  26,   1,  85},
    {  41,  10,  67, 138,  77, 110,  90,  47, 114},
    { 101,  29,  16,  10,  85, 128, 101, 196,  26},
    {  57,  18,  10, 102, 102, 213,  34,  20,  43},
    { 117,  20,  15,  36, 163, 128,  68,   1,  26}
  },
  {
    { 138,  31,  36, 171,  27, 166,  38,  44, 229},
    {  67,  87,  58, 169,  82, 115,  26,  59, 179},
    {  63,  59,  90, 180,  59, 166,  93,  73, 154},
    {  40,  40,  21, 116, 143, 209,  34,  39, 175},
    {  57,  46,  22,  24, 128,   1,  54,  17,  37},
    {  47,  15,  16, 183,  34, 223,  49,  45, 183},
    {  46,  17,  33, 183,   6,  98,  15,  32, 183},
    {  65,  32,  73, 115,  28, 128,  23, 128, 205},
    {  40,   3,   9, 115,  51, 192,  18,   6, 223},
    {  87,  37,   9, 115,  59,  77,  64,  21,  47}
  },
  {
    { 104,  55,  44, 218,   9,  54,  53, 130, 226},
    {  64,  90,  70, 205,  40,  41,  23,  26,  57},
    {  54,  57, 112, 184,   5,  41,  38, 166, 213},
    {  30,  34,  26, 133, 152, 116,  10,  32, 134},
    {  75,  32,  12,  51, 192, 255, 160,  43,  51},
    {  39,  19,  53, 221,  26, 114,  32,  73, 255},
    {  31,   9,  65, 234,   2,  15,   1, 118,  73},
    {  88,  31,  35,  67, 102,  85,  55, 186,  85},
    {  56,  21,  23, 111,  59, 205,  45,  37, 192},
    {  55,  38,  70, 124,  73, 102,   1,  34,  98}
  },
  {
    { 102,  61,  71,  37,  34,  53,  31, 243, 192},
    {  69,  60,  71,  38,  73, 119,  28, 222,  37},
    {  68,  45, 128,  34,   1,  47,  11, 245, 171},
    {  62,  17,  19,  70, 146,  85,  55,  62,  70},
    {  75,  15,   9,   9,  64, 255, 184, 119,  16},
    {  37,  43,  37, 154, 100, 163,  85, 160,   1},
    {  63,   9,  92, 136,  28,  64,  32, 201,  85},
    {  86,   6,  28,   5,  64, 255,  25, 248,   1},
    {  56,   8,  17, 132, 137, 255,  55, 116, 128},
    {  58,  15,  20,  82, 135,  57,  26, 121,  40}
  },
```

```
  {
    { 164,  50,  31, 137, 154, 133,  25,  35, 218},
    {  51, 103,  44, 131, 131, 123,  31,   6, 158},
    {  86,  40,  64, 135, 148, 224,  45, 183, 128},
    {  22,  26,  17, 131, 240, 154,  14,   1, 209},
    {  83,  12,  13,  54, 192, 255,  68,  47,  28},
    {  45,  16,  21,  91,  64, 222,   7,   1, 197},
    {  56,  21,  39, 155,  60, 138,  23, 102, 213},
    {  85,  26,  85,  85, 128, 128,  32, 146, 171},
    {  18,  11,   7,  63, 144, 171,   4,   4, 246},
    {  35,  27,  10, 146, 174, 171,  12,  26, 128}
  },
  {
    { 190,  80,  35,  99, 180,  80, 126,  54,  45},
    {  85, 126,  47,  87, 176,  51,  41,  20,  32},
    { 101,  75, 128, 139, 118, 146, 116, 128,  85},
    {  56,  41,  15, 176, 236,  85,  37,   9,  62},
    { 146,  36,  19,  30, 171, 255,  97,  27,  20},
    {  71,  30,  17, 119, 118, 255,  17,  18, 138},
    { 101,  38,  60, 138,  55,  70,  43,  26, 142},
    { 138,  45,  61,  62, 219,   1,  81, 188,  64},
    {  32,  41,  20, 117, 151, 142,  20,  21, 163},
    { 112,  19,  12,  61, 195, 128,  48,   4,  24}
  }
};
```

Chapter 12: Intraframe Prediction

Intraframe prediction uses already-coded macroblocks within the current frame to approximate the contents of the current macroblock. It applies to intra-coded macroblocks in an interframe and to all macroblocks in a key frame.

Relative to the current macroblock “M”, the already-coded macroblocks include all macroblocks above M together with the macroblocks on the same row as, and to the left of, M, though at most four of these macroblocks are actually used: the block “A” directly above M, the blocks immediately to the left and right of A, and the block immediately to the left of M.

Each of the prediction modes (i.e., means of extrapolation from already-calculated values) uses fairly simple arithmetic on pixel values whose positions, relative to the current position, are defined by the mode.

The chroma (U and V) and luma (Y) predictions are independent of each other.

The relative addressing of pixels applied to macroblocks on the upper row or left column of the frame will sometimes cause pixels outside the visible frame to be referenced. Usually such out-of-bounds pixels have an assumed value of `129` for pixels to the left of the leftmost column of the visible frame and `127` for pixels above the top row of the visible frame (including the special case of the pixel above and to the left of the top-left pixel in the visible frame). Exceptions to this (associated to certain modes) will be noted below.

The already-coded macroblocks referenced by intra-prediction have been “reconstructed”, that is, have been predicted and residue-adjusted (as described in Chapter 14), but have not been loop-filtered. While it does process the edges between individual macroblocks and individual subblocks, loop filtering (described in Chapter 15) is applied to the frame as a whole, after all of the macroblocks have been reconstructed.

12.1 mb_skip_coeff

The single bool flag is decoded using `prob_skip_false` if and only if `mb_no_coeff_skip` is set to `1` (see Sections 9.10 and 9.11). If `mb_no_coeff_skip` is set to `0` then this value defaults to `0`.

12.2 Chroma Prediction

The chroma prediction is a little simpler than the luma prediction, so we treat it first. Each of the chroma modes treats U and V identically, that is, the U and V prediction values are calculated in parallel, using the same relative addressing and arithmetic in each of the two planes.

The modes extrapolate prediction values using the 8-pixel row “A” lying immediately above the block (that is, the bottom chroma row of the macroblock immediately above the current macroblock) and the 8-pixel column “L” immediately to the left of the block (that is, the rightmost chroma column of the macroblock immediately to the left of the current macroblock).

Vertical prediction (chroma mode `V_PRED`) simply fills each 8-pixel row of the 8x8 chroma block with a copy of the “above” row (A). If the current macroblock lies on the top row of the frame, all 8 of the pixel values in A are assigned the value `127`.

Similarly, horizontal prediction (`H_PRED`) fills each 8-pixel column of the 8x8 chroma block with a copy of the “left” column (L). If the current macroblock is in the left column of the frame, all 8 pixel values in L are assigned the value `129`.

DC prediction (`DC_PRED`) fills the 8x8 chroma block with a single value. In the generic case of a macroblock lying below the top row and right of the leftmost column of the frame, this value is the average of the 16 (genuinely visible) pixels in the (union of the) above row A and left column L.

Otherwise, if the current macroblock lies on the top row of the frame, the average of the 8 pixels in L is used; if it lies in the left column of the frame, the average of the 8 pixels in A is used. Note that the averages used in these exceptional cases are not the same as those that would be arrived at by using the out-of-bounds A and L values defined for V_PRED and H_PRED. In the case of the leftmost macroblock on the top row of the frame the 8x8 block is simply filled with the constant value `128` .

For `DC_PRED` , apart from the exceptional case of the top left macroblock, we are averaging either 16 or 8 pixel values to get a single prediction value that fills the 8x8 block. The rounding is done as follows:

```
int sum;    /* sum of 8 or 16 pixels at (at least) 16-bit precision */
int shf;    /* base 2 logarithm of the number of pixels (3 or 4) */

Pixel DCvalue = (sum + (1 << (shf-1))) >> shf;
```

Because the summands are all valid pixels, no “clamp” is necessary in the calculation of `DCvalue` .

The remaining “True Motion” (`TM_PRED`) chroma mode gets its name from an older technique of video compression used by On2 Technologies, to which it bears some relation. In addition to the row “A” and column “L”, `TM_PRED` uses the pixel “P” above and to the left of the chroma block.

In this mode, we propagate the horizontal differences between pixels in A (starting from P), using the pixels from L to start each row. The exact algorithm is as follows.

```
void TMpred(
    Pixel b[8][8],      /* chroma (U or V) prediction block */
    const Pixel A[8],   /* row of already-constructed pixels above block */
    const Pixel L[8],   /* column of "" just to the left of block */
    const Pixel P       /* pixel just to the left of A and above L*/
) {
    int r = 0;          /* row */
    do {
        int c = 0;      /* column */
        do {
            b[r][c] = clamp255( L[r]+ A[c] - P);
        } while( ++c < 8);
    } while( ++r < 8);
}
```

Note that the process could equivalently be described as propagating the vertical differences between pixels in L (starting from P), using the pixels from A to start each column.

An implementation of chroma intra-prediction may be found in the reference decoder file `reconintra.c` .

Unlike `DC_PRED`, for macroblocks on the top row or left edge `TM_PRED` does use the out-of-bounds values of `127` and `129` (respectively) defined for `V_PRED` and `H_PRED`.

12.3 Luma Prediction

The prediction processes for the first four 16x16 luma modes (`DC_PRED`, `V_PRED`, `H_PRED`, and `TM_PRED`) are essentially identical to the corresponding chroma prediction processes described above, the only difference being that we are predicting a single 16x16 luma block instead of two 8x8 chroma blocks.

Thus, the row “A” and column “L” here contain 16 pixels, the DC prediction is calculated using 16 or 32 pixels (and shf is 4 or 5), and we of course fill the entire prediction buffer, that is, 16 rows (or columns) containing 16 pixels each. The reference implementation of 16x16 luma prediction is also in `reconintra.c`.

In the remaining luma mode (`B_PRED`), each 4x4 Y subblock is independently predicted using one of ten modes (listed, along with their encodings, in Chapter 11).

Also, unlike the full-macroblock modes already described, some of the subblock modes use prediction pixels above and to the right of the current subblock. In detail, each 4x4 subblock “B” is predicted using (at most) the 4-pixel column “L” immediately to the left of B and the 8-pixel row “A” immediately above B, consisting of the 4 pixels above B followed by the 4 adjacent pixels above and to the right of B, together with the single pixel “P” immediately to the left of A (and immediately above L).

For the purpose of subblock intra-prediction, the pixels immediately to the left and right of a pixel in a subblock are the same as the pixels immediately to the left and right of the corresponding pixel in the frame buffer “F”. Vertical offsets behave similarly: The above row A lies immediately above B in F, and the adjacent pixels in the left column L are separated by a single row in F.

Because entire macroblocks (as opposed to their constituent subblocks) are reconstructed in raster-scan order, for subblocks lying along the right edge (and not along the top row) of the current macroblock, the four “extra” prediction pixels in A above and to the right of B have not yet actually been constructed.

Subblocks 7, 11, and 15 are affected. All three of these subblocks use the same extra pixels as does subblock 3 (at the upper right corner of the macroblock), namely the 4 pixels immediately above and to the right of subblock 3. Writing `(R,C)` for a frame buffer position offset from the upper left corner of the current macroblock by R rows and C columns, the extra pixels for all the right-edge subblocks (3, 7, 11, and 15) are at positions (-1,16), (-1,17), (-1,18), and (-1,19).

The details of the prediction modes are most easily described in code.

```
/* Result pixels are often averages of two or three predictor pixels.
   The following subroutines are used to calculate these averages.
   Because the arguments are valid pixels, no clamping is necessary.
   An actual implementation would probably use inline functions or macros. */

/* Compute weighted average centered at y w/adjacent x, z */

Pixel avg3( Pixel x, Pixel y, Pixel z) { return (x + y + y + z + 2) >> 2;}

/* Weighted average of 3 adjacent pixels centered at p */
```

```
Pixel avg3p( const Pixel *p) { return avg3( p[-1], p[0], p[1]);}

/* Simple average of x and y */

Pixel avg2( Pixel x, Pixel y) { return (x + y + 1) >> 1;}

/* Average of p[0] and p[1] may be considered to be a synthetic pixel
   lying between the two, that is, one half-step past p. */

Pixel avg2p( const Pixel *p) { return avg2( p[0], p[1]);}

/* Main function.  Out-of-frame pixels in A or L should be set to 128. */

void subblock_intra_predict(
    Pixel B[4][4],      /* Y subblock prediction buffer */
    const Pixel *A,     /* A[0]...A[7] = above row, A[-1] = P */
    const Pixel *L,     /* L[0]...L[3] = left column, L[-1] = P */
    intra_bmode mode     /* enum is in section 11.1 above */
) {
    Pixel E[9];         /* 9 already-constructed edge pixels */
    E[0] = L[3];  E[1] = L[2];  E[2] = L[1];  E[3] = L[0];
    E[4] = A[-1];       /* == L[-1] == P */
    E[5] = A[0];  E[6] = A[1];  E[7] = A[2];  E[8] = A[3];

  switch( mode) {
    /* First four modes are similar to corresponding full-block modes. */

    case B_DC_PRED: /* like 16x16 DC_PRED except out-of-frame */
    {               /* prediction pixels are always taken to be 128 */
        int v = 4;      /* DC sum/avg, 4 is rounding adjustment */
        int i = 0;  do { v += A[i] + L[i];}  while( ++i < 4);
        v >>= 3;        /* averaging 8 pixels */
        i = 0;  do {    /* fill prediction buffer with constant DC value */
            int j = 0;  do { B[i][j] = v;}  while( ++j < 4);
        } while( ++i < 4);
        break;
    }

    case B_TM_PRED: /* just like 16x16 TM_PRED */
    {
        int r = 0;  do {
            int c = 0;  do {
                B[r][c] = clamp255( L[r] + A[c] - A[-1]);
            } while( ++c < 4);
        } while( ++r < 4);
        break;
    }

    case B_VE_PRED: /* like 16x16 V_PRED except using averages */
    {
        int c = 0;  do { /* all 4 rows = smoothed top row */
            B[0][c] = B[1][c] = B[2][c] = B[3][c] = avg3p( A + c);
        } while( ++c < 4);
        break;
    }
```

```
    case B_HE_PRED: /* like 16x16 H_PRED except using averages */
    {
        /* Bottom row is exceptional because L[4] does not exist */
        int v = avg3( L[2], L[3], L[3]);
        int r = 3;  while( 1) { /* all 4 columns = smoothed left column */
            B[r][0] = B[r][1] = B[r][2] = B[r][3] = v;
            if( --r < 0)
                break;
            v = avg3p( L + r);  /* upper 3 rows use average of 3 pixels */
        }
        break;
    }

    /* The remaining six "diagonal" modes subdivide the prediction buffer into
       diagonal lines.  All the pixels on each line are assigned the same
       value; this value is (a smoothed or synthetic version of) an
       already-constructed predictor value lying on the same line.  For
       clarity, in the comments, we express the positions of these predictor
       pixels relative to the upper left corner of the destination array B.

       These modes are unique to subblock prediction and have no full-block
       analogues.  The first two use lines at +|- 45 degrees from horizontal
       (or, equivalently, vertical), that is, lines whose slopes are +|- 1. */

    case B_LD_PRED: /* southwest (left and down) step = (-1, 1) or (1,-1) */
        /* avg3p( A + j) is the "smoothed" pixel at (-1,j) */
        B[0][0] = avg3p( A + 1);
        B[0][1] = B[1][0] = avg3p( A + 2);
        B[0][2] = B[1][1] = B[2][0] = avg3p( A + 3);
        B[0][3] = B[1][2] = B[2][1] = B[3][0] = avg3p( A + 4);
        B[1][3] = B[2][2] = B[3][1] = avg3p( A + 5);
        B[2][3] = B[3][2] = avg3p( A + 6);
        B[3][3] = avg3( A[6], A[7], A[7]);  /* A[8] does not exist */
        break;

    case B_RD_PRED: /* southeast (right and down) step = (1,1) or (-1,-1) */
        B[3][0] = avg3p( E + 1);  /* predictor is from (2, -1) */
        B[3][1] = B[2][0] = avg3p( E + 2);  /* (1, -1) */
        B[3][2] = B[2][1] = B[1][0] = avg3p( E + 3);  /* (0, -1) */
        B[3][3] = B[2][2] = B[1][1] = B[0][0] = avg3p( E + 4);  /* (-1, -1) */
        B[2][3] = B[1][2] = B[0][1] = avg3p( E + 5);  /* (-1, 0) */
        B[1][3] = B[0][2] = avg3p( E + 6);  /* (-1, 1) */
        B[0][3] = avg3p( E + 7);  /* (-1, 2) */
        break;

    /* The remaining 4 diagonal modes use lines whose slopes are +|- 2
       and +|- 1/2.  The angles of these lines are roughly +|- 27 degrees
       from horizontal or vertical.

       Unlike the 45 degree diagonals, here we often need to "synthesize"
       predictor pixels midway between two actual predictors using avg2p(p),
       which we think of as returning the pixel "at" p[1/2]. */

    case B_VR_PRED: /* SSE (vertical right) step = (2,1) or (-2,-1) */
        B[3][0] = avg3p( E + 2);  /* predictor is from (1, -1) */
```

```
        B[2][0] = avg3p( E + 3);  /* (0, -1) */
        B[3][1] = B[1][0] = avg3p( E + 4);  /* (-1,  -1) */
        B[2][1] = B[0][0] = avg2p( E + 4);  /* (-1, -1/2) */
        B[3][2] = B[1][1] = avg3p( E + 5);  /* (-1,   0) */
        B[2][2] = B[0][1] = avg2p( E + 5);  /* (-1, 1/2) */
        B[3][3] = B[1][2] = avg3p( E + 6);  /* (-1,   1) */
        B[2][3] = B[0][2] = avg2p( E + 6);  /* (-1, 3/2) */
        B[1][3] = avg3p( E + 7);  /* (-1, 2) */
        B[0][3] = avg2p( E + 7);  /* (-1, 5/2) */
        break;

    case B_VL_PRED: /* SSW (vertical left) step = (2,-1) or (-2,1) */
        B[0][0] = avg2p( A);  /* predictor is from (-1, 1/2) */
        B[1][0] = avg3p( A + 1);  /* (-1, 1) */
        B[2][0] = B[0][1] = avg2p( A + 1);  /* (-1, 3/2) */
        B[1][1] = B[3][0] = avg3p( A + 2);  /* (-1,   2) */
        B[2][1] = B[0][2] = avg2p( A + 2);  /* (-1, 5/2) */
        B[3][1] = B[1][2] = avg3p( A + 3);  /* (-1,   3) */
        B[2][2] = B[0][3] = avg2p( A + 3);  /* (-1, 7/2) */
        B[3][2] = B[1][3] = avg3p( A + 4);  /* (-1,   4) */
        /* Last two values do not strictly follow the pattern. */
        B[2][3] = avg3p( A + 5);  /* (-1, 5) [avg2p( A + 4) = (-1,9/2)] */
        B[3][3] = avg3p( A + 6);  /* (-1, 6) [avg3p( A + 5) = (-1,5)] */
        break;

    case B_HD_PRED: /* ESE (horizontal down) step = (1,2) or (-1,-2) */
        B[3][0] = avg2p( E);  /* predictor is from (5/2, -1) */
        B[3][1] = avg3p( E + 1);  /* (2, -1) */
        B[2][0] = B[3][2] = svg2p( E + 1);  /* ( 3/2, -1) */
        B[2][1] = B[3][3] = avg3p( E + 2);  /* (   1, -1) */
        B[2][2] = B[1][0] = avg2p( E + 2);  /* ( 1/2, -1) */
        B[2][3] = B[1][1] = avg3p( E + 3);  /* (   0, -1) */
        B[1][2] = B[0][0] = avg2p( E + 3);  /* (-1/2, -1) */
        B[1][3] = B[0][1] = avg3p( E + 4);  /* (  -1, -1) */
        B[0][2] = avg3p( E + 5);  /* (-1, 0) */
        B[0][3] = avg3p( E + 6);  /* (-1, 1) */
        break;

    case B_HU_PRED: /* ENE (horizontal up) step = (1,-2) or (-1,2) */
        B[0][0] = avg2p( L);  /* predictor is from ( 1/2, -1) */
        B[0][1] = avg3p( L + 1);  /* ( 1, -1) */
        B[0][2] = B[1][0] = avg2p( L + 1);  /* (3/2, -1) */
        B[0][3] = B[1][1] = avg3p( L + 2);  /* (  2, -1) */
        B[1][2] = B[2][0] = avg2p( L + 2);  /* (5/2, -1) */
        B[1][3] = B[2][1] = avg3( L[2], L[3], L[3]);  /* ( 3, -1) */
        /* Not possible to follow pattern for much of the bottom row
           because no (nearby) already-constructed pixels lie on the
           diagonals in question. */
        B[2][2] = B[2][3] = B[3][0] = B[3][1] = B[3][2] = B[3][3] = L[3];
  }
}
```

The reference decoder implementation of subblock intra-prediction may be found in `reconintra4x4.c`.

Chapter 13: DCT Coefficient Decoding

The second data partition consists of an encoding of the quantized DCT (and WHT) coefficients of the residue signal. As discussed in the format overview (Chapter 2), for each macroblock, the residue is added to the (intra- or inter-generated) prediction buffer to produce the final (except for loop-filtering) reconstructed macroblock.

VP8 works exclusively with 4x4 DCTs and WHTs, applied to the 24 (or 25 with the Y2 subblock) 4x4 subblocks of a macroblock. The ordering of macroblocks within any of the “residue” partitions in general follows the same raster-scan as used in the first “prediction” partition.

For all intra- and inter-prediction modes apart from `B_PRED` (intra: whose Y subblocks are independently predicted) and `SPLIT_MV` (inter) each macroblock’s residue record begins with the Y2 component of the residue, coded using a WHT. `B_PRED` and `SPLIT_MV` coded macroblocks omit this WHT, instead specifying the 0^{th} DCT coefficient of each of the 16 Y subblocks as part of its DCT.

After the optional Y2 block, the residue record continues with 16 DCTs for the Y subblocks, followed by 4 DCTs for the U subblocks, ending with 4 DCTs for the V subblocks. The subblocks occur in the usual order.

The DCTs and WHT are tree-coded using a 12-element alphabet whose members we call tokens. Except for the end of block token (which sets the remaining subblock coefficients to zero and is followed by the next block), each token (sometimes augmented with data immediately following the token) specifies the value of the single coefficient at the current (implicit) position and is followed by a token applying to the next (implicit) position.

For all the Y and chroma subblocks, the ordering of the coefficients follows a so-called zig-zag order. DCTs begin at coefficient 1 if Y2 is present, and begin at coefficient 0 if Y2 is absent. The WHT for a Y2 subblock always begins at coefficient 0.

13.1 MB Without non-Zero Coefficient Values

If the flag within macroblock mode info indicates that a macroblock does not have any non-zero coefficients, the decoding process of DCT coefficients is skipped for the macroblock.

13.2 Coding of Individual Coefficient Values

The coding of coefficient tokens is the same for the DCT and WHT and for the remainder of this chapter DCT should be taken to mean either DCT or WHT.

All tokens (except end-of-block) specify either a single unsigned value or a range of unsigned values (immediately) followed by a simple probabilistic encoding of the offset of the value from the base of that range.

Non-zero values (of either type) are then followed by a flag indicating the sign of the coded value (negative if `1`, positive if `0`).

Here are the tokens and decoding tree.

```
typedef enum
{
    DCT_0,      /* value 0 */
    DCT_1,      /* 1 */
    DCT_2,      /* 2 */
    DCT_3,      /* 3 */
    DCT_4,      /* 4 */
    dct_cat1,   /* range 5 - 6   (size 2) */
    dct_cat2,   /* 7 - 10    (4) */
    dct_cat3,   /* 11 - 18   (8) */
    dct_cat4,   /* 19 - 34   (16) */
    dct_cat5,   /* 35 - 66   (32) */
    dct_cat6,   /* 67 - 2048   (1982) */
    dct_eob,    /* end of block */

    num_dct_tokens   /* 12 */
}
dct_token;

const tree_index coef_tree [2 * (num_dct_tokens - 1)] =
{
 -dct_eob, 2,               /* eob = "0"    */
  -DCT_0, 4,                /* 0    = "10"  */
   -DCT_1, 6,               /* 1    = "110" */
    8, 12,
     -DCT_2, 10,            /* 2    = "11100" */
      -DCT_3, -DCT_4,       /* 3    = "111010", 4 = "111011" */
     14, 16,
      -dct_cat1, -dct_cat2, /* cat1 =  "111100", cat2 = "111101" */
     18, 20,
      -dct_cat3, -dct_cat4, /* cat3 = "1111100", cat4 = "1111101" */
      -dct_cat5, -dct_cat6  /* cat4 = "1111110", cat4 = "1111111" */
};
```

While in general all DCT coefficients are decoded using the same tree, decoding of certain DCT coefficients may skip the first branch, whose preceding coefficient is a `DCT_0`. This makes use of the fact that in any block last non zero coefficient before the end of the block is not `0`, therefore no `dct_eob` follows a `DCT_0` coefficient in any block.

The tokens `dct_cat1` ... `dct_cat6` specify ranges of unsigned values, the value within the range being formed by adding an unsigned offset (whose width is 1, 2, 3, 4, 5, or 11 bits, respectively) to the base of the range, using the following algorithm and fixed probability tables.

```
uint DCTextra( bool_decoder *d, const Prob *p)
{
    uint v = 0;
    do { v += v + read_bool( d, *p);}  while( *++p);
    return v;
}
```

```
const Prob Pcat1[] = { 159, 0};
const Prob Pcat2[] = { 165, 145, 0};
const Prob Pcat3[] = { 173, 148, 140, 0};
const Prob Pcat4[] = { 176, 155, 140, 135, 0};
const Prob Pcat5[] = { 180, 157, 141, 134, 130, 0};
const Prob Pcat6[] =
    { 254, 254, 243, 230, 196, 177, 153, 140, 133, 130, 129, 0};
```

If `v`, the unsigned value decoded using the coefficient tree, possibly augmented by the process above, is non-zero, its sign is set by simply reading a flag:

```
if( read_bool( d, 128))
    v = -v;
```

13.3 Token Probabilities

The probability specification for the token tree (unlike that for the “extra bits” described above) is rather involved. It uses three pieces of context to index a large probability table, the contents of which may be incrementally modified in the frame header. The full (non-constant) probability table is laid out as follows.

```
Prob coef_probs [4] [8] [3] [num_dct_tokens-1];
```

Working from the outside in, the outermost dimension is indexed by the type of plane being decoded:

- `0` - Y beginning at coefficient 1 (i.e., Y after Y2)
- `1` - Y2
- `2` - U or V
- `3` - Y beginning at coefficient 0 (i.e., Y in the absence of Y2).

The next dimension is selected by the position of the coefficient being decoded. That position c steps by ones up to 15, starting from zero for block types 1, 2, or 3 and starting from one for block type 0. The second array index is then

```
coef_bands [c]
```

where

```
const int coef_bands [16] = {
     0, 1, 2, 3, 6, 4, 5, 6, 6, 6, 6, 6, 6, 6, 7
};
```

is a fixed mapping of position to “band”.

The third dimension is the trickiest. Roughly speaking, it measures the “local complexity” or extent to which nearby coefficients are non-zero.

For the first coefficient (DC, unless the block type is 0), we consider the (already encoded) blocks within the same plane (Y2, Y, U, or V) above and to the left of the current block. The context index is then the number (0, 1 or 2) of these blocks that had at least one non-zero coefficient in their residue record.

Beyond the first coefficient, the context index is determined by the absolute value of the most recently decoded coefficient (necessarily within the current block) and is `0` if the last coefficient was a zero, `1` if it was plus or minus one, and `2` if its absolute value exceeded one.

Note that the intuitive meaning of this measure changes as coefficients are decoded. For example, prior to the first token, a zero means that the neighbors are empty, suggesting that the current block may also be empty. After the first token, because an end-of-block token must have at least one non-zero value before it, a zero means that we just decoded a zero and hence guarantees that a non-zero coefficient will appear later in this block. However, this shift in meaning is perfectly okay because the complete context depends also on the coefficient band (and since band 0 is occupied exclusively by position 0).

As with other contexts used by VP8, the “neighboring block” context described here needs a special definition for subblocks lying along the top row or left edge of the frame. These “non-existent” predictors above and to the left of the image are simply taken to be empty — that is, taken to contain no non-zero coefficients.

The residue decoding of each macroblock then requires, in each of two directions (above and to the left), an aggregate coefficient predictor consisting of a single Y2 predictor, two predictors for each of U and V, and four predictors for Y. In accordance with the scan-ordering of macroblocks, a decoder needs to maintain a single “left” aggregate predictor and a row of “above” aggregate predictors.

Before decoding any residue, these maintained predictors may simply be cleared, in compliance with the definition of “non-existent” prediction. After each block is decoded, the two predictors referenced by the block are replaced with the (empty or non-empty) state of the block, in preparation for the later decoding of the blocks below and to the right of the block just decoded.

The fourth, and final, dimension of the token probability array is of course indexed by (half) the position in the token tree structure, as are all tree probability arrays.

While we have in fact completely described the coefficient decoding procedure, the reader will probably find it helpful to consult the reference implementation, which can be found in the file `detokenize.c`.

13.4 Token Probability Updates

As mentioned above, the token-decoding probabilities may change from frame to frame. After detection of a key frame, they are of course set to their defaults shown in Section 13.5; this must occur before decoding the remainder of the header, as both key frames and interframes may adjust these probabilities.

The layout and semantics of the coefficient probability update record (Section I of the frame header) are straightforward. For each position in the `coef_probs` array there occurs a fixed-probability bool indicating whether or not the corresponding probability should be updated. If the bool is true, there follows a `P(8)` replacing that probability. Note that updates are cumulative, that is, a probability updated on one frame is in effect for all ensuing frames until the next key frame, or until the probability is explicitly updated by another frame.

The algorithm to effect the foregoing is simple:

```
int i = 0;  do {
 int j = 0;  do {
  int k = 0;  do {
   int t = 0;  do {

        if( read_bool( d, coef_update_probs [i] [j] [k] [t]))
            coef_probs [i] [j] [k] [t] = read_literal( d, 8);

   } while( ++t < num_dct_tokens - 1);
  } while( ++k < 3);
 } while( ++j < 8);
} while( ++i < 4);
```

The (constant) update probabilities are as follows (they may also be found in the reference decoder file `coef_update_probs.c`).

```
const Prob coef_update_probs [4] [8] [3] [num_dct_tokens-1] =
{
 {
  {
   { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
   { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
   { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
  },
  {
   { 176, 246, 255, 255, 255, 255, 255, 255, 255, 255, 255},
   { 223, 241, 252, 255, 255, 255, 255, 255, 255, 255, 255},
   { 249, 253, 253, 255, 255, 255, 255, 255, 255, 255, 255}
  },
  {
   { 255, 244, 252, 255, 255, 255, 255, 255, 255, 255, 255},
   { 234, 254, 254, 255, 255, 255, 255, 255, 255, 255, 255},
   { 253, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
  },
  {
   { 255, 246, 254, 255, 255, 255, 255, 255, 255, 255, 255},
   { 239, 253, 254, 255, 255, 255, 255, 255, 255, 255, 255},
   { 254, 255, 254, 255, 255, 255, 255, 255, 255, 255, 255}
  },
  {
   { 255, 248, 254, 255, 255, 255, 255, 255, 255, 255, 255},
   { 251, 255, 254, 255, 255, 255, 255, 255, 255, 255, 255},
   { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
  },
  {
   { 255, 253, 254, 255, 255, 255, 255, 255, 255, 255, 255},
   { 251, 254, 254, 255, 255, 255, 255, 255, 255, 255, 255},
   { 254, 255, 254, 255, 255, 255, 255, 255, 255, 255, 255}
  },
  {
   { 255, 254, 253, 255, 254, 255, 255, 255, 255, 255, 255},
   { 250, 255, 254, 255, 254, 255, 255, 255, 255, 255, 255},
```

```
    { 254, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   }
  },
  {
   {
    { 217, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 225, 252, 241, 253, 255, 255, 254, 255, 255, 255, 255},
    { 234, 250, 241, 250, 253, 255, 253, 254, 255, 255, 255}
   },
   {
    { 255, 254, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 223, 254, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 238, 253, 254, 254, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 248, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 249, 254, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 253, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 247, 254, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 253, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 252, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 254, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 253, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 254, 253, 255, 255, 255, 255, 255, 255, 255, 255},
    { 250, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 254, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   }
  },
  {
   {
    { 186, 251, 250, 255, 255, 255, 255, 255, 255, 255, 255},
    { 234, 251, 244, 254, 255, 255, 255, 255, 255, 255, 255},
    { 251, 251, 243, 253, 254, 255, 254, 255, 255, 255, 255}
   },
   {
```

```
    { 255, 253, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 236, 253, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 251, 253, 253, 254, 254, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 254, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 254, 254, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 254, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 254, 254, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 254, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 254, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   }
  },
  {
   {
    { 248, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 250, 254, 252, 254, 255, 255, 255, 255, 255, 255, 255},
    { 248, 254, 249, 253, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 253, 253, 255, 255, 255, 255, 255, 255, 255, 255},
    { 246, 253, 253, 255, 255, 255, 255, 255, 255, 255, 255},
    { 252, 254, 251, 254, 254, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 254, 252, 255, 255, 255, 255, 255, 255, 255, 255},
    { 248, 254, 253, 255, 255, 255, 255, 255, 255, 255, 255},
    { 253, 255, 254, 254, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 251, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 245, 251, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 253, 253, 254, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
```

```
    { 255, 251, 253, 255, 255, 255, 255, 255, 255, 255, 255},
    { 252, 253, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 254, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 252, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 249, 255, 254, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 254, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 255, 253, 255, 255, 255, 255, 255, 255, 255, 255},
    { 250, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   },
   {
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 254, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255},
    { 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255}
   }
  }
};
```

13.5 Default Token Probability Table

The default token probabilities are as follows.

```
const Prob default_coef_probs [4] [8] [3] [num_dct_tokens - 1] =
{
 {
  {
   { 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128},
   { 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128},
   { 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128}
  },
  {
   { 253, 136, 254, 255, 228, 219, 128, 128, 128, 128, 128},
   { 189, 129, 242, 255, 227, 213, 255, 219, 128, 128, 128},
   { 106, 126, 227, 252, 214, 209, 255, 255, 128, 128, 128}
  },
  {
   {   1,  98, 248, 255, 236, 226, 255, 255, 128, 128, 128},
   { 181, 133, 238, 254, 221, 234, 255, 154, 128, 128, 128},
   {  78, 134, 202, 247, 198, 180, 255, 219, 128, 128, 128}
  },
  {
   {   1, 185, 249, 255, 243, 255, 128, 128, 128, 128, 128},
   { 184, 150, 247, 255, 236, 224, 128, 128, 128, 128, 128},
   {  77, 110, 216, 255, 236, 230, 128, 128, 128, 128, 128}
  },
  {
   {   1, 101, 251, 255, 241, 255, 128, 128, 128, 128, 128},
   { 170, 139, 241, 252, 236, 209, 255, 255, 128, 128, 128},
   {  37, 116, 196, 243, 228, 255, 255, 255, 128, 128, 128}
  },
  {
```

```
    {   1, 204, 254, 255, 245, 255, 128, 128, 128, 128, 128},
    { 207, 160, 250, 255, 238, 128, 128, 128, 128, 128, 128},
    { 102, 103, 231, 255, 211, 171, 128, 128, 128, 128, 128}
   },
   {
    {   1, 152, 252, 255, 240, 255, 128, 128, 128, 128, 128},
    { 177, 135, 243, 255, 234, 225, 128, 128, 128, 128, 128},
    {  80, 129, 211, 255, 194, 224, 128, 128, 128, 128, 128}
   },
   {
    {   1,   1, 255, 128, 128, 128, 128, 128, 128, 128, 128},
    { 246,   1, 255, 128, 128, 128, 128, 128, 128, 128, 128},
    { 255, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128}
   }
  },
  {
   {
    { 198,  35, 237, 223, 193, 187, 162, 160, 145, 155,  62},
    { 131,  45, 198, 221, 172, 176, 220, 157, 252, 221,   1},
    {  68,  47, 146, 208, 149, 167, 221, 162, 255, 223, 128}
   },
   {
    {   1, 149, 241, 255, 221, 224, 255, 255, 128, 128, 128},
    { 184, 141, 234, 253, 222, 220, 255, 199, 128, 128, 128},
    {  81,  99, 181, 242, 176, 190, 249, 202, 255, 255, 128}
   },
   {
    {   1, 129, 232, 253, 214, 197, 242, 196, 255, 255, 128},
    {  99, 121, 210, 250, 201, 198, 255, 202, 128, 128, 128},
    {  23,  91, 163, 242, 170, 187, 247, 210, 255, 255, 128}
   },
   {
    {   1, 200, 246, 255, 234, 255, 128, 128, 128, 128, 128},
    { 109, 178, 241, 255, 231, 245, 255, 255, 128, 128, 128},
    {  44, 130, 201, 253, 205, 192, 255, 255, 128, 128, 128}
   },
   {
    {   1, 132, 239, 251, 219, 209, 255, 165, 128, 128, 128},
    {  94, 136, 225, 251, 218, 190, 255, 255, 128, 128, 128},
    {  22, 100, 174, 245, 186, 161, 255, 199, 128, 128, 128}
   },
   {
    {   1, 182, 249, 255, 232, 235, 128, 128, 128, 128, 128},
    { 124, 143, 241, 255, 227, 234, 128, 128, 128, 128, 128},
    {  35,  77, 181, 251, 193, 211, 255, 205, 128, 128, 128}
   },
   {
    {   1, 157, 247, 255, 236, 231, 255, 255, 128, 128, 128},
    { 121, 141, 235, 255, 225, 227, 255, 255, 128, 128, 128},
    {  45,  99, 188, 251, 195, 217, 255, 224, 128, 128, 128}
   },
   {
    {   1,   1, 251, 255, 213, 255, 128, 128, 128, 128, 128},
    { 203,   1, 248, 255, 255, 128, 128, 128, 128, 128, 128},
    { 137,   1, 177, 255, 224, 255, 128, 128, 128, 128, 128}
   }
  },
```

```
{
 {
  { 253,   9, 248, 251, 207, 208, 255, 192, 128, 128, 128},
  { 175,  13, 224, 243, 193, 185, 249, 198, 255, 255, 128},
  {  73,  17, 171, 221, 161, 179, 236, 167, 255, 234, 128}
 },
 {
  {   1,  95, 247, 253, 212, 183, 255, 255, 128, 128, 128},
  { 239,  90, 244, 250, 211, 209, 255, 255, 128, 128, 128},
  { 155,  77, 195, 248, 188, 195, 255, 255, 128, 128, 128}
 },
 {
  {   1,  24, 239, 251, 218, 219, 255, 205, 128, 128, 128},
  { 201,  51, 219, 255, 196, 186, 128, 128, 128, 128, 128},
  {  69,  46, 190, 239, 201, 218, 255, 228, 128, 128, 128}
 },
 {
  {   1, 191, 251, 255, 255, 128, 128, 128, 128, 128, 128},
  { 223, 165, 249, 255, 213, 255, 128, 128, 128, 128, 128},
  { 141, 124, 248, 255, 255, 128, 128, 128, 128, 128, 128}
 },
 {
  {   1,  16, 248, 255, 255, 128, 128, 128, 128, 128, 128},
  { 190,  36, 230, 255, 236, 255, 128, 128, 128, 128, 128},
  { 149,   1, 255, 128, 128, 128, 128, 128, 128, 128, 128}
 },
 {
  {   1, 226, 255, 128, 128, 128, 128, 128, 128, 128, 128},
  { 247, 192, 255, 128, 128, 128, 128, 128, 128, 128, 128},
  { 240, 128, 255, 128, 128, 128, 128, 128, 128, 128, 128}
 },
 {
  {   1, 134, 252, 255, 255, 128, 128, 128, 128, 128, 128},
  { 213,  62, 250, 255, 255, 128, 128, 128, 128, 128, 128},
  {  55,  93, 255, 128, 128, 128, 128, 128, 128, 128, 128}
 },
 {
  { 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128},
  { 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128},
  { 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128}
 }
},
{
 {
  { 202,  24, 213, 235, 186, 191, 220, 160, 240, 175, 255},
  { 126,  38, 182, 232, 169, 184, 228, 174, 255, 187, 128},
  {  61,  46, 138, 219, 151, 178, 240, 170, 255, 216, 128}
 },
 {
  {   1, 112, 230, 250, 199, 191, 247, 159, 255, 255, 128},
  { 166, 109, 228, 252, 211, 215, 255, 174, 128, 128, 128},
  {  39,  77, 162, 232, 172, 180, 245, 178, 255, 255, 128}
 },
 {
  {   1,  52, 220, 246, 198, 199, 249, 220, 255, 255, 128},
  { 124,  74, 191, 243, 183, 193, 250, 221, 255, 255, 128},
  {  24,  71, 130, 219, 154, 170, 243, 182, 255, 255, 128}
```

```
    },
    {
     {   1, 182, 225, 249, 219, 240, 255, 224, 128, 128, 128},
     { 149, 150, 226, 252, 216, 205, 255, 171, 128, 128, 128},
     {  28, 108, 170, 242, 183, 194, 254, 223, 255, 255, 128}
    },
    {
     {   1,  81, 230, 252, 204, 203, 255, 192, 128, 128, 128},
     { 123, 102, 209, 247, 188, 196, 255, 233, 128, 128, 128},
     {  20,  95, 153, 243, 164, 173, 255, 203, 128, 128, 128}
    },
    {
     {   1, 222, 248, 255, 216, 213, 128, 128, 128, 128, 128},
     { 168, 175, 246, 252, 235, 205, 255, 255, 128, 128, 128},
     {  47, 116, 215, 255, 211, 212, 255, 255, 128, 128, 128}
    },
    {
     {   1, 121, 236, 253, 212, 214, 255, 255, 128, 128, 128},
     { 141,  84, 213, 252, 201, 202, 255, 219, 128, 128, 128},
     {  42,  80, 160, 240, 162, 185, 255, 205, 128, 128, 128}
    },
    {
     {   1,   1, 255, 128, 128, 128, 128, 128, 128, 128, 128},
     { 244,   1, 255, 128, 128, 128, 128, 128, 128, 128, 128},
     { 238,   1, 255, 128, 128, 128, 128, 128, 128, 128, 128}
    }
   }
};
```

Chapter 14: DCT and WHT Inversion and Macroblock Reconstruction

14.1 Dequantization

After decoding the DCTs/WHTs as described above, each (quantized) coefficient in each subblock is multiplied by one of six dequantization factors, the choice of factor depending on the plane (Y2, Y, or chroma) and position (DC = coefficient zero, AC = any other coefficient). If the current macroblock has overridden the quantization level (as described in Chapter 10) then the six factors are looked up from two dequantization tables with appropriate scaling and clamping using the single index supplied by the override. Otherwise, the frame-level dequantization factors (as described in Section 9.6 are used. In either case, the multiplies are computed and stored using 16-bit signed integers.

The two dequantization tables, which may also be found in the reference decoder file `quant_common.c`, are as follows.

```
static const int dc_qlookup[QINDEX_RANGE] =
{
    4,   5,   6,   7,   8,   9,  10,  10,  11,  12,  13,  14,  15,  16,  17,  17,
   18,  19,  20,  20,  21,  21,  22,  22,  23,  23,  24,  25,  25,  26,  27,  28,
   29,  30,  31,  32,  33,  34,  35,  36,  37,  37,  38,  39,  40,  41,  42,  43,
   44,  45,  46,  46,  47,  48,  49,  50,  51,  52,  53,  54,  55,  56,  57,  58,
   59,  60,  61,  62,  63,  64,  65,  66,  67,  68,  69,  70,  71,  72,  73,  74,
   75,  76,  76,  77,  78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,
   91,  93,  95,  96,  98, 100, 101, 102, 104, 106, 108, 110, 112, 114, 116, 118,
  122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 143, 145, 148, 151, 154, 157,
};

static const int ac_qlookup[QINDEX_RANGE] =
{
    4,   5,   6,   7,   8,   9,  10,  11,  12,  13,  14,  15,  16,  17,  18,  19,
   20,  21,  22,  23,  24,  25,  26,  27,  28,  29,  30,  31,  32,  33,  34,  35,
   36,  37,  38,  39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
   52,  53,  54,  55,  56,  57,  58,  60,  62,  64,  66,  68,  70,  72,  74,  76,
   78,  80,  82,  84,  86,  88,  90,  92,  94,  96,  98, 100, 102, 104, 106, 108,
  110, 112, 114, 116, 119, 122, 125, 128, 131, 134, 137, 140, 143, 146, 149, 152,
  155, 158, 161, 164, 167, 170, 173, 177, 181, 185, 189, 193, 197, 201, 205, 209,
  213, 217, 221, 225, 229, 234, 239, 245, 249, 254, 259, 264, 269, 274, 279, 284,
};
```

Lookup values from the above two tables are directly used the DC and AC coefficients in Y1 respectively. For Y2 and chroma, values from above tables undergo either a scaling process or clamping processing before the multiplies. Details to these scaling and clamping can be found related lookup functions in `quant_common.c`.

14.2 Inverse Transforms

If the Y2 residue block exists (i.e., the macroblock luma mode is not `SPLITMV` or `B_PRED`), it is inverted first (using the inverse WHT) and the element of the result at row i, column j is used as the 0^{th} coefficient of the Y subblock at position (i, j), that is, the Y subblock whose index is (i * 4) + j. As discussed in Chapter 13, if the luma mode is `B_PRED` or `SPLITMV`, the 0^{th} Y coefficients are part of the residue signal for the subblocks themselves.

In either case, the inverse transforms for the sixteen Y subblocks and eight chroma subblocks are computed next. All 24 of these inversions are independent of each other; their results may (at least conceptually) be stored in 24 separate 4x4 arrays.

As is done by the reference decoder, an implementation may wish to represent the prediction and residue buffers as macroblock-sized arrays (that is, a 16x16 Y buffer and two 8x8 chroma buffers). Regarding the inverse DCT implementation given below, this requires a simple adjustment to the address calculation for the resulting residue pixels.

14.3 Implementation of the WHT Inversion

As previously discussed (see Chapters 2 and 13), for macroblocks encoded using prediction modes other than `B_PRED` and `SPLITMV`, the DC values derived from the DCT transform on the 16 Y blocks are collected to construct a 25th block of a macroblock(16 Y, 4 U, 4 V constitute the 24 blocks). This 25th block is transformed using a Walsh-Hadamard transform (WHT).

The inputs to the inverse WHT (that is, the dequantized coefficients), the intermediate “horizontally detransformed” signal, and the completely detransformed residue signal are all stored as arrays of 16-bit signed integers.

Following the tradition of specifying bitstream format using the decoding process, we specify the inverse WHT in the decoding process using the following C style source code:

```
void vp8_short_inv_walsh4x4_c(short *input, short *output)
{
  int i;
  int a1, b1, c1, d1;
  int a2, b2, c2, d2;
  short *ip = input;
  short *op = output;
  int temp1, temp2;

  for(i=0;i<4;i++)
  {
    a1 = ip[0] + ip[12];
    b1 = ip[4] + ip[8];
    c1 = ip[4] - ip[8];
    d1 = ip[0] - ip[12];

    op[0] = a1 + b1;
    op[4] = c1 + d1;
    op[8] = a1 - b1;
    op[12]= d1 - c1;
    ip++;
    op++;
  }
  ip = output;
  op = output;
  for(i=0;i<4;i++)
  {
    a1 = ip[0] + ip[3];
    b1 = ip[1] + ip[2];
    c1 = ip[1] - ip[2];
    d1 = ip[0] - ip[3];
```

```
    a2 = a1 + b1;
    b2 = c1 + d1;
    c2 = a1 - b1;
    d2 = d1 - c1;

    op[0] = (a2+3)>>3;
    op[1] = (b2+3)>>3;
    op[2] = (c2+3)>>3;
    op[3] = (d2+3)>>3;

    ip+=4;
    op+=4;
  }
}
```

In the case that there is only one non-zero DC value in input, the inverse transform can be simplified to the following:

```
void vp8_short_inv_walsh4x4_1_c(short *input, short *output)
{
  int i;
  int a1;
  short *op=output;



  a1 = ((input[0] + 3)>>3);

  for(i=0;i<4;i++)
  {
    op[0] = a1;
    op[1] = a1;
    op[2] = a1;
    op[3] = a1;
    op+=4;
  }
}
```

It should be noted, a conforming decoder should implement the inverse transform using exactly the same rounding to achieve bit-wise matching output to the output of the process specified by the above “C” source code.

The reference decoder WHT inversion may be found in the files `invtrans.c` and `idctllm.c`.

14.4 Implementation of the DCT Inversion

All of the DCT inversions are computed in exactly the same way. In principle, VP8 uses a classical 2D inverse discrete cosine transform, implemented as two passes of 1-D inverse DCT. The 1-D inverse DCT was calculated using a similar algorithm to what was described in the paper “Practical Fast 1-D DCT Algorithms with 11 Multiplications” by Loeffler, Lightenberg and Moschytz. However, the paper only provided the 8-point and 16-point version of the algorithms, which was adapted by On2 to perform the 4-point 1-D DCT.

Accurate calculation of 1-D DCT of the above algorithm requires infinite precision. VP8 of course can use only a finite-precision approximation. Also, the inverse DCT used by VP8 takes care of normalization of the standard unitary transform, that is, every dequantized coefficient has roughly double the size of the corresponding unitary coefficient. However, at all but the highest datarates, the discrepancy between transmitted and ideal coefficients is due almost entirely to (lossy) compression and not to errors induced by finite-precision arithmetic.

The inputs to the inverse DCT (that is, the dequantized coefficients), the intermediate “horizontally detransformed” signal, and the completely detransformed residue signal are all stored as arrays of 16-bit signed integers. The details of the computation are as follows.

It should also be noted that this implementation makes use of 16-bit fixed point version of two multiplication constants:

$$\sqrt{2} \cdot \cos\left(\frac{\pi}{8}\right)$$

$$\sqrt{2} \cdot \sin\left(\frac{\pi}{8}\right)$$

Because the first constant is bigger than 1, to maintain the same 16-bit fixed point precision as the second one, we make use of the fact that

$$x \cdot a = x + x \cdot (a\text{-}1)$$

therefore

$$x \cdot \sqrt{2} \cdot \cos\left(\frac{\pi}{8}\right) = x + x \cdot \left(\sqrt{2} \cdot \cos\left(\frac{\pi}{8}\right)\text{-}1\right)$$

```
/* IDCT implementation */
static const int cospi8sqrt2minus1=20091;
static const int sinpi8sqrt2      =35468;
void short_idct4x4llm_c(short *input, short *output, int pitch)
{
  int i;
  int a1, b1, c1, d1;

  short *ip=input;
  short *op=output;
  int temp1, temp2;
  int shortpitch = pitch>>1;

  for(i=0;i<4;i++)
  {
    a1 = ip[0]+ip[8];
    b1 = ip[0]-ip[8];

    temp1 = (ip[4] * sinpi8sqrt2)>>16;
    temp2 = ip[12]+((ip[12] * cospi8sqrt2minus1)>>16);
    c1 = temp1 - temp2;
```

```
    temp1 = ip[4] + ((ip[4] * cospi8sqrt2minus1)>>16);
    temp2 = (ip[12] * sinpi8sqrt2)>>16;
    d1 = temp1 + temp2;

    op[shortpitch*0] = a1+d1;
    op[shortpitch*3] = a1-d1;
    op[shortpitch*1] = b1+c1;
    op[shortpitch*2] = b1-c1;

    ip++;
    op++;
  }
  ip = output;
  op = output;
  for(i=0;i<4;i++)
  {
    a1 = ip[0]+ip[2];
    b1 = ip[0]-ip[2];

    temp1 = (ip[1] * sinpi8sqrt2)>>16;
    temp2 = ip[3]+((ip[3] * cospi8sqrt2minus1)>>16);
    c1 = temp1 - temp2;

    temp1 = ip[1] + ((ip[1] * cospi8sqrt2minus1)>>16);
    temp2 = (ip[3] * sinpi8sqrt2)>>16;
    d1 = temp1 + temp2;

    op[0] = (a1+d1+4)>>3;
    op[3] = (a1-d1+4)>>3;
    op[1] = (b1+c1+4)>>3;
    op[2] = (b1-c1+4)>>3;

    ip+=shortpitch;
    op+=shortpitch;
  }
}
```

The reference decoder DCT inversion may be found in the files `invtrans.c` and `idctllm.c`.

14.5 Summation of Predictor and Residue

Finally, the prediction and residue signals are summed to form the reconstructed macroblock, which, except for loop filtering (taken up next), completes the decoding process.

The summing procedure is fairly straightforward, having only a couple of details. The prediction and residue buffers are both arrays of 16-bit signed integers. Each individual (Y, U, and V pixel) result is calculated first as a 32-bit sum of the prediction and residue, and is then saturated to 8-bit unsigned range (using, say, the `clamp255` function defined above) before being stored as an 8-bit unsigned pixel value.

VP8 also supports a mode where the encoding of a bitstream guarantees all reconstructed pixel values between 0 and 255, compliant bitstreams of such requirements have the `clamp_type` bit in the frame header set to 1. In such case, the `clamp255` is no longer required.

The summation process is the same, regardless of the (intra or inter) mode of prediction in effect for the macroblock. The reference decoder implementation of reconstruction may be found in the file `recon.c`.

Chapter 15: Loop Filter

Loop filtering is the last stage of frame reconstruction and the next-to-last stage of the decoding process. The loop filter is applied to the entire frame after the summation of predictor and residue described in Chapter 14.

The purpose of the loop filter is to eliminate (or at least reduce) visually objectionable artifacts associated with the semi-independence of the coding of macroblocks and their constituent subblocks.

As was discussed in Chapter 5, the loop filter is “integral” to decoding, in that the results of loop filtering are used in the prediction of subsequent frames. Consequently, a functional decoder implementation must perform loop filtering exactly as described here. This is in distinction to any postprocessing that may be applied only to the image immediately before display; such postprocessing is entirely at the option of the implementor (and/or user) and has no effect on decoding per se.

The baseline frame level parameters controlling the loop filter are defined in the frame header (Chapter 9.4) along with a mechanism for adjustment based on a macroblock’s prediction mode and/or reference frame. The first is a flag selecting the type of filter (normal or simple), the other two are numbers (`loop_filter_level` and `sharpness_level`) that adjust the strength or sensitivity of the filter. As described in Chapters 9.3 and 10, `loop_filter_level` may be also overridden on a per-macroblock basis using segmentation.

Loop filtering is one of the more computationally-intensive aspects of VP8 decoding. This is the reason for the existence of the optional less-demanding simple filter type. Also, the loop filter is completely disabled if the loop_filter_level in the frame header is zero; macroblock-level overrides are ignored in this case. (It is of course possible for a compressor to encode a frame in which only a few macroblocks are loop filtered: The global loop_filter_level must be non-zero and each macroblock can select one of four levels, most of which could be zero.)

To facilitate efficient implementation, the VP8 decoding algorithms generally, and the loop filter especially, were designed with SIMD (“Single Instruction Multiple Datum” or “integer vector”) processors in mind. The reference decoder implementation of loop filtering (found in `loopfilter.c`) is, in effect, a portable SIMD specification of the loop filtering algorithms intended to simplify a realization on an actual SIMD processor.

Unfortunately, the approach taken there does not lead to maximal efficency (restricted to the C language, that is) and, as far as a pure algorithm specification is concerned, is in places obscure. For example, various aspects of filtering are conditioned on absolute differences lying below certain thresholds. An ordinary C implementation would simply discriminate amongst these behaviors using if statements. The reference decoder instead effects this by “masking arithmetic”, that is, using “and” operations to (conditionally) zero-out values to be added or subtracted to pixels. Furthermore, the structure holding the various threshold values is artificially parallelized. While this mimics closely the approach taken in vector-processor machine language, it is not how one usually programs in C.

In this document, we take a different approach and present the algorithms in a more straightforward, idiomatic, and terse C style. Together with the reference version, we hope to provide the “best of both worlds”, that is, a pure algorithm specification here and a strong suggestion as to an optimal actual implementation in `loopfilter.c` .

We begin by discussing the aspects of loop filtering that are independent of the controlling parameters and type of filter chosen.

15.1 Filter Geometry and Overall Procedure

The Y, U, and V planes are processed independently and, except for the values of certain control parameters (derived from the `loop_filter_level` and `sharpness_level`), identically.

The loop filter acts on the edges between adjacent macroblocks and on the edges between adjacent subblocks of a macroblock. All such edges are horizontal or vertical. For each pixel position on an edge, a small number (two or three) of pixels adjacent to either side of the position are examined and possibly modified. The displacements of these pixels are at a right angle to the edge orientation, that is, for a horizontal edge, we treat the pixels immediately above and below the edge position, for a vertical edge, we treat the pixels immediately to the left and right of the edge.

We call this collection of pixels associated to an edge position a segment; the length of a segment is 2, 4, 6, or 8. Excepting that the normal filter uses slightly different algorithms for, and that either filter may apply different control parameters to, the edges between macroblocks and those between subblocks, the treatment of edges is quite uniform: All segments straddling an edge are treated identically, there is no distinction between the treatment of horizontal and vertical edges, whether between macroblocks or between subblocks.

As a consequence, adjacent subblock edges within a macroblock may be concatenated and processed in their entirety. There is a single 8-pixel long vertical edge horizontally centered in each of the U and V blocks (the concatenation of upper and lower 4-pixel edges between chroma subblocks), and three 16-pixel long vertical edges at horizontal positions 1/4, 1/2, and 3/4 the width of the luma macroblock, each representing the concatenation of four 4-pixel sub-edges between pairs of Y subblocks.

The macroblocks comprising the frame are processed in the usual raster-scan order. Each macroblock is “responsible for” the inter-macroblock edges immediately above and left of it (but not the edges below and right of it), as well as the edges between its subblocks.

For each macroblock M, there are four filtering steps, which are, (almost) in order:

1. If M is not on the leftmost column of macroblocks, filter across the left (vertical) inter-macroblock edge of M.

2. Filter across the vertical subblock edges within M.

3. If M is not on the topmost row of macroblocks, filter across the top (horizontal) inter-macroblock edge of M.

4. Filter across the horizontal subblock edges within M.

We write MY, MU, and MV for the planar constituents of M, that is, the 16x16 luma block, 8x8 U block, and 8x8 V block comprising M.

In step 1, for each of the three blocks MY, MU, and MV, we filter each of the (16 luma or 8 chroma) segments straddling the column separating the block from the block immediately to the left of it, using the inter-macroblock filter and controls associated to the `loop_filter_level` and `sharpness_level`.

In step 4, we filter across the (three luma and one each for U and V) vertical subblock edges described above, this time using the inter-subblock filter and controls.

Step 2 and 4 are skipped for macroblocks that satisfy the following two conditions:

1. Macroblock coding mode is neither `B_PRED` nor `SPLTMV`;
2. There is no DCT coefficient coded for the whole macroblock.

For these macroblocks, loop filtering for edges between subblocks internal to a macroblock is effectively skipped. This skip strategy significantly reduces VP8 loop-filtering complexity.

Edges between macroblocks and those between subblocks are treated with different control parameters (and, in the case of the normal filter, with different algorithms); luma and chroma edges are also treated with different control parameters. Except for pixel addressing, there is no distinction between the treatment of vertical and horizontal edges. Luma edges are always 16 pixels long, chroma edges are always 8 pixels long, and the segments straddling an edge are treated identically; this of course facilitates vector processing.

Because many pixels belong to segments straddling two or more edges, and so will be filtered more than once, the order in which edges are processed given above must be respected by any implementation. Within a single edge, however, the segments straddling that edge are disjoint and the order in which these segments are processed is immaterial.

Before taking up the filtering algorithms themselves, we should emphasize a point already made: Even though the pixel segments associated to a macroblock are antecedent to the macroblock (that is, lie within the macroblock or in already-constructed macroblocks), a macroblock must not be filtered immediately after its “reconstruction” (described in Chapter 14). Rather, the loop filter applies after all the macroblocks have been “reconstructed” (i.e., had their predictor summed with their residue); correct decoding is predicated on the fact that already-constructed portions of the current frame referenced via intra-prediction (described in Chapter 12) are not yet filtered.

15.2 Simple Filter

Having described the overall procedure of, and pixels affected by, the loop filter, we turn our attention to the treatment of individual segments straddling edges. We begin by describing the simple filter, which, as the reader might guess, is somewhat simpler than the normal filter.

Roughly speaking, the idea of loop filtering is, within limits, to reduce the difference between pixels straddling an edge. Differences in excess of a threshold (associated to the `loop_filter_level`) are assumed to be “natural” and are unmodified; differences below the threshold are assumed to be artifacts of quantization and the (partially) separate coding of blocks, and are reduced via the procedures described below. While the `loop_filter_level` is in principle arbitrary, the levels chosen by a VP8 compressor tend to be correlated to quantization levels.

Most of the filtering arithmetic is done using 8-bit signed operands (having a range -128 to +127, inclusive), supplemented by 16-bit temporaries holding results of multiplies.

Sums and other temporaries need to be “clamped” to a valid signed 8-bit range:

```
int8 c( int v)
{
```

```
    return (int8) (v < -128 ? -128 : (v < 128 ? v : 127));
}
```

Since pixel values themselves are unsigned 8-bit numbers, we need to convert between signed and unsigned values:

```
/* Convert pixel value (0 <= v <= 255) to an 8-bit signed number. */
int8 u2s( Pixel v) { return (int8) (v - 128);}

/* Clamp, then convert signed number back to pixel value. */
Pixel s2u( int v) { return (Pixel) ( c(v) + 128);}
```

Filtering is often predicated on absolute-value thresholds. The following function is the equivalent of the standard library function `abs`, whose prototype is found in the standard header file `stdlib.h`. For us, the argument `v` is always the difference between two pixels and lies in the range $-255 \le v \le +255$.

```
int abs( int v) { return v < 0?  -v : v;}
```

An actual implementation would of course use inline functions or macros to accomplish these trivial procedures (which are used by both the normal and simple loop filters). An optimal implementation would probably express them in machine language, perhaps using SIMD vector instructions. On many SIMD processors, the saturation accomplished by the above clamping function is often folded into the arithmetic instructions themselves, obviating the explicit step taken here.

To simplify the specification of relative pixel positions, we use the word before to mean “immediately above” (for a vertical segment straddling a horizontal edge) or “immediately to the left of” (for a horizontal segment straddling a vertical edge) and the word after to mean “immediately below” or “immediately to the right of”.

Given an edge, a segment, and a limit value, the simple loop filter computes a value based on the four pixels that straddle the edge (two either side). If that value is below a supplied limit, then, very roughly speaking, the two pixel values are brought closer to each other, “shaving off” something like a quarter of the difference. The same procedure is used for all segments straddling any type of edge, regardless of the nature (inter-macroblock, inter-subblock, luma, or chroma) of the edge; only the limit value depends on the edge-type.

The exact procedure (for a single segment) is as follows; the subroutine common_adjust is used by both the simple filter presented here and the normal filters discussed in Section 15.3.

```
int8 common_adjust(
    int use_outer_taps,   /* filter is 2 or 4 taps wide */
    const Pixel *P1,     /* pixel before P0 */
    Pixel *P0,           /* pixel before edge */
    Pixel *Q0,           /* pixel after edge */
    const Pixel *Q1      /* pixel after Q0 */
) {
    cint8 p1 = u2s( *P1);   /* retrieve and convert all 4 pixels */
```

```
    cint8 p0 = u2s( *P0);
    cint8 q0 = u2s( *Q0);
    cint8 q1 = u2s( *Q1);

    /* Disregarding clamping, when "use_outer_taps" is false, "a" is 3*(q0-p0).
       Since we are about to divide "a" by 8, in this case we end up
       multiplying the edge difference by 5/8.

       When "use_outer_taps" is true (as for the simple filter),
       "a" is p1 - 3*p0 + 3*q0 - q1, which can be thought of as a refinement
       of 2*(q0 - p0) and the adjustment is something like (q0 - p0)/4. */

    int8 a = c( ( use_outer_taps? c(p1 - q1) : 0 ) + 3*(q0 - p0) );

    /* b is used to balance the rounding of a/8 in the case where
       the "fractional" part "f" of a/8 is exactly 1/2. */

    cint8 b = (a & 7)==4 ? -1 : 0;

    /* Divide a by 8, rounding up when f >= 1/2.
       Although not strictly part of the "C" language,
       the right-shift is assumed to propagate the sign bit. */

    a = c( a + 4) >> 3;

    /* Subtract "a" from q0, "bringing it closer" to p0. */

    *Q0 = s2u( q0 - a);

    /* Add "a" (with adjustment "b") to p0, "bringing it closer" to q0.

       The clamp of "a+b", while present in the reference decoder, is
       superfluous; we have -16 <= a <= 15 at this point. */

    *P0 = s2u( p0 + c( a + b));

    return a;
}
```

```
void simple_segment(
    uint8 edge_limit,    /* do nothing if edge difference exceeds limit */
    const Pixel *P1,     /* pixel before P0 */
    Pixel *P0,           /* pixel before edge */
    Pixel *Q0,           /* pixel after edge */
    const Pixel *Q1      /* pixel after Q0 */
) {
    if( (abs(*P0 - *Q0)*2 + abs(P1-Q1)/2) <= edge_limit))
        common_adjust( 1, P1, P0, Q0, Q1);   /* use outer taps */
}
```

We make a couple of remarks about the rounding procedure above. When `b` is zero (that is, when the “fractional part” of `a` is not 1/2), we are (except for clamping) adding the same number to `p0` as we are

subtracting from `q0`. This preserves the average value of `p0` and `q0` but the resulting difference between `p0` and `q0` is always even; in particular, the smallest non-zero gradation ± 1 is not possible here.

When `b` is one, the value we add to `p0` (again except for clamping) is one less than the value we are subtracting from `q0`. In this case, the resulting difference is always odd (and the small gradation ± 1 is possible) but the average value is reduced by 1/2, yielding, for instance, a very slight darkening in the luma plane. (In the very unlikely event of appreciable darkening after a large number of interframes, a compressor would of course eventually compensate for this in the selection of predictor and/or residue.)

The derivation of the `edge_limit` value used above, which depends on the `loop_filter_level` and `sharpness_level`, as well as the type of edge being processed, will be taken up after we describe the normal loop filtering algorithm below.

15.3 Normal Filter

The normal loop filter is a refinement of the simple loop filter; all of the general discussion above applies here as well. In particular, the functions `c`, `u2s`, `s2u`, `abs`, and `common_adjust` are used by both the normal and simple filters.

As mentioned above, the normal algorithms for inter-macroblock and inter-subblock edges differ. Nonetheless, they have a great deal in common: They use similar threshold algorithms to disable the filter and to detect high internal edge variance (which influences the filtering algorithm). Both algorithms also use, at least conditionally, the simple filter adjustment procedure described above.

The common thresholding algorithms are as follows.

```
/* All functions take (among other things) a segment (of length at most
   4 + 4 = 8) symmetrically straddling an edge.

   The pixel values (or pointers) are always given in order, from the
   "beforemost" to the "aftermost". So, for a horizontal edge (written "|"),
   an 8-pixel segment would be ordered p3 p2 p1 p0 | q0 q1 q2 q3. */


/* Filtering is disabled if the difference between any two adjacent "interior"
   pixels in the 8-pixel segment exceeds the relevant threshold (I). A more
   complex thresholding calculation is done for the group of four pixels that
   straddle the edge, in line with the calculation in simple_segment() above. */

int filter_yes(
    uint8 I,        /* limit on interior differences */
    uint8 E,        /* limit at the edge */

    cint8 p3, cint8 p2, cint8 p1, cint8 p0, /* pixels before edge */
    cint8 q0, cint8 q1, cint8 q2, cint8 q3  /* pixels after edge */
) {
    return  (abs(p0 - q0)*2 + abs(p1-q1)/2) <= E
        &&  abs(p3 - p2) <= I  &&  abs(p2 - p1) <= I  &&  abs(p1 - p0) <= I
        &&  abs(q3 - q2) <= I  &&  abs(q2 - q1) <= I  &&  abs(q1 - q0) <= I;
}
```

```
/* Filtering is altered if (at least) one of the differences on either
   side of the edge exceeds a threshold (we have "high edge variance"). */

int hev(
    uint8 threshold,
    cint8 p1, cint8 p0, /* pixels before edge */
    cint8 q0, cint8 q1  /* pixels after edge */
) {
    return abs(p1 - p0) > threshold  ||  abs(q1 - q0) > threshold;
}
```

The subblock filter is a variant of the simple filter. In fact, if we have high edge variance, the adjustment is exactly as for the simple filter. Otherwise, the simple adjustment (without outer taps) is applied and the two pixels one step in from the edge pixels are adjusted by roughly half the amount by which the two edge pixels are adjusted; since the edge adjustment here is essentially 3/8 the edge difference, the inner adjustment is approximately 3/16 the edge difference.

```
void subblock_filter(
    uint8 hev_threshold,     /* detect high edge variance */
    uint8 interior_limit,    /* possibly disable filter */
    uint8 edge_limit,
    cint8 *P3, cint8 *P2, int8 *P1, int8 *P0,   /* pixels before edge */
    int8 *Q0, int8 *Q1, cint8 *Q2, cint8 *Q3    /* pixels after edge */
```

```
) {
    cint8 p3 = u2s(*P3), p2 = u2s(*P2), p1 = u2s(*P1), p0 = u2s(*P0);
    cint8 q0 = u2s(*Q0), q1 = u2s(*Q1), q2 = u2s(*Q2), q3 = u2s(*Q3);

    if( filter_yes( interior_limit, edge_limit, q3, q2, q1, q0, p0, p1, p2, p3))
    {
        const int hv = hev( hev_threshold, p1, p0, q0, q1);

        cint8 a = ( common_adjust( hv, P1, P0, Q0, Q1) + 1) >> 1;

        if( !hv) {
            *Q1 = s2u( q1 - a);
            *P1 = s2u( p1 + a);
        }
    }
}
```

The inter-macroblock filter has potentially wider scope. If the edge variance is high, it performs the simple adjustment (using the outer taps, just like the simple filter and the corresponding case of the normal subblock filter). If the edge variance is low, we begin with the same basic filter calculation and apply multiples of it to pixel pairs symmetric about the edge; the magnitude of adjustment decays as we move away from the edge and six of the pixels in the segment are affected.

```
void MBfilter(
    uint8 hev_threshold,     /* detect high edge variance */
    uint8 interior_limit,    /* possibly disable filter */
    uint8 edge_limit,
    cint8 *P3, int8 *P2, int8 *P1, int8 *P0,  /* pixels before edge */
    int8 *Q0, int8 *Q1, int8 *Q2, cint8 *Q3   /* pixels after edge */
) {
    cint8 p3 = u2s(*P3), p2 = u2s(*P2), p1 = u2s(*P1), p0 = u2s(*P0);
    cint8 q0 = u2s(*Q0), q1 = u2s(*Q1), q2 = u2s(*Q2), q3 = u2s(*Q3);

    if( filter_yes( interior_limit, edge_limit, q3, q2, q1, q0, p0, p1, p2, p3))
    {
        if( !hev( hev_threshold, p1, p0, q0, q1))
        {
            /* Same as the initial calculation in "common_adjust",
               w is something like twice the edge difference */

            const int8 w = c( c(p1 - q1) + 3*(q0 - p0) );

            /* 9/64 is approximately 9/63 = 1/7 and 1<<7 = 128 = 2*64.
               So this a, used to adjust the pixels adjacent to the edge,
               is something like 3/7 the edge difference. */

            int8 a = c( 27*w + 63) >> 7;

            *Q0 = s2u( q0 - a);  *P0 = s2u( p0 + a);

            /* Next two are adjusted by 2/7 the edge difference */

            a = c( 18*w + 63) >> 7;
```

```
            *Q1 = s2u( q1 - a);  *P1 = s2u( p1 + a);

            /* Last two are adjusted by 1/7 the edge difference */

            a = c( 9*w + 63) >> 7;

            *Q2 = s2u( q2 - a);  *P2 = s2u( p2 + a);

        } else                                  /* if hev, do simple filter */
            common_adjust( 1, P1, P0, Q0, Q1);   /* using outer taps */
    }

}
```

15.4 Calculation of Control Parameters

We conclude the discussion of loop filtering by showing how the thresholds supplied to the procedures above are derived from the two control parameters `sharpness_level` (an unsigned 3-bit number having maximum value 7) and `loop_filter_level` (an unsigned 6-bit number having maximum value 63).

While the `sharpness_level` is constant over the frame, individual macroblocks may override the `loop_filter_level` with one of four possibilities supplied in the frame header (as described in Chapter 10).

Both the simple and normal filters disable filtering if a value derived from the four pixels that straddle the edge (2 either side) exceeds a threshold / limit value.

```
/* Luma and Chroma use the same inter-macroblock edge limit */
uint8 mbedge_limit = ((loop_filter_level + 2) * 2) + interior_limit;

/* Luma and Chroma use the same inter-subblock edge limit */
uint8 sub_bedge_limit = (loop_filter_level * 2) + interior_limit;
```

The remaining thresholds are used only by the normal filters. The filter-disabling interior difference limit is the same for all edges (luma, chroma, inter-subblock, inter-macroblock) and is given by the following.

```
uint8 interior_limit = loop_filter_level;

if( sharpness_level)
{
    interior_limit  >>=  sharpness_level > 4 ?  2 : 1;
    if( interior_limit > 9 - sharpness_level)
        interior_limit = 9 - sharpness_level;
}
if( !interior_limit)
    interior_limit = 1;
```

Finally, we give the derivation of the high edge-variance threshold, which is also the same for all edge types.

```
uint8 hev_threshold = 0;

if( we_are_decoding_akey_frame)          /* current frame is a key frame */
{
    if( loop_filter_level >= 40)
        hev_threshold = 2;
    else if( loop_filter_level >= 15)
        hev_threshold = 1;
}
else                                    /* current frame is an interframe */
{
    if( loop_filter_level >= 40)
        hev_threshold = 3;
    else if( loop_filter_level >= 20)
        hev_threshold = 2;
    else if( loop_filter_level >= 15)
        hev_threshold = 1;
}
```

Chapter 16: Interframe Macroblock Prediction Records

We describe the layout and semantics of the prediction records for macroblocks in an interframe.

After the feature specification (which is described in Chapter 10 and is identical for intraframes and interframes), there comes a Bool(prob_intra), which indicates inter-prediction (i.e., prediction from prior frames) when true and intra-prediction (i.e., prediction from already-coded portions of the current frame) when false. The zero-probability `prob_intra` is set by field J of the frame header.

16.1 Intra-Predicted Macroblocks

For intra-prediction, the layout of the prediction data is essentially the same as the layout for key frames, although the contexts used by the decoding process are slightly different.

As discussed in Chapter 8, the “outer” Y mode here uses a different tree from that used in key frames, repeated here for convenience.

```
const tree_index ymode_tree [2 * (num_ymodes - 1)] =
{
 -DC_PRED, 2,            /* root: DC_PRED = "0", "1" subtree */
  4, 6,                  /* "1" subtree has 2 descendant subtrees */
   -V_PRED, -H_PRED,     /* "10" subtree:  V_PRED = "100", H_PRED = "101" */
   -TM_PRED, -B_PRED     /* "11" subtree:  TM_PRED = "110", B_PRED = "111" */
};
```

The probability table used to decode this tree is variable. As described in Chapter 9, it (along with the similarly-treated UV table) can be updated by field J of the frame header. Similar to the coefficient-decoding probabilities, such updates are cumulative and affect all ensuing frames until the next key frame or explicit update. The default probabilities for the Y and UV tables are

```
Prob ymode_prob [num_ymodes - 1] = { 112, 86, 140, 37};
Prob uv_mode_prob [num_uv_modes - 1] = { 162, 101, 204};
```

These defaults must be restored after detection of a key frame.

Just as for key frames, if the Y mode is `B_PRED`, there next comes an encoding of the `intra_bpred` mode used by each of the sixteen Y subblocks. These encodings use the same tree as does that for key frames but, in place of the contexts used in key frames, use the single fixed probability table

```
const Prob bmode_prob [num_intra_bmodes - 1] = {
    120, 90, 79, 133, 87, 85, 80, 111, 151
};
```

Last comes the chroma mode, again coded using the same tree as that for key frames, this time using the dynamic `uv_mode_prob` table described above.

The calculation of the intra-prediction buffer is identical to that described for key frames in Chapter 12.

16.2 Inter-Predicted Macroblocks

Otherwise (when the above bool is true), we are using inter-prediction (which of course only happens for interframes), to which we now restrict our attention.

The next datum is then another bool, `B( prob_last)`, selecting the reference frame. If 0, the reference frame is previous frame (last frame); if 1, another bool (`prob_gf`) selects the reference frame between golden frame (0) or altref frame (1). The probabilities `prob_last` and `prob_gf` are set in field J of the frame header.

Together with setting the reference frame, the purpose of inter-mode decoding is to set a motion vector for each of the sixteen Y subblocks of the current macroblock. This then defines the calculation of the inter-prediction buffer (detailed in Chapter 18). While the net effect of inter-mode decoding is straightforward, the implementation is somewhat complex; the (lossless) compression achieved by this method justifies the complexity.

After the reference frame selector comes the mode (or motion vector reference) applied to the macroblock as a whole, coded using the following enumeration and tree. Setting `mv_nearest = num_ymodes` is a convenience that allows a single variable to unambiguously hold an inter- or intraprediction mode.

```
typedef enum
{
    mv_nearest = num_ymodes, /* use "nearest" motion vector for entire MB */
    mv_near,                 /* use "next nearest" "" */
    mv_zero,                 /* use zero "" */
    mv_new,                  /* use explicit offset from implicit "" */
    mv_split,                /* use multiple motion vectors */

    num_mv_refs = mv_split + 1 - mv_nearest
}
mv_ref;

const tree_index mv_ref_tree [2 * (num_mv_refs - 1)] =
{
 -mv_zero, 2,                /* zero = "0" */
  -mv_nearest, 4,            /* nearest = "10" */
   -mv_near, 6,              /* near = "110" */
     -mv_new, -mv_split      /* new = "1110", split = "1111" */
};
```

16.3 Mode and Motion Vector Contexts

The probability table used to decode the `mv_ref`, along with three reference motion vectors used by the selected mode, is calculated via a survey of the already-decoded motion vectors in (up to) 3 nearby macroblocks.

The algorithm generates a sorted list of distinct motion vectors adjacent to the search site. The `best_mv` is the vector with the highest score. The `nearest_mv` is the non-zero vector with the highest score. The `near_mv` is the non-zero vector with the next highest score. The number of motion vectors coded using the

`SPLITMV` mode is scored using the same weighting and is returned with the scores of the best, nearest, and near vectors.

The three adjacent macroblocks above, left, and above-left are considered in order. If the macroblock is intra-coded, no action is taken. Otherwise, the motion vector is compared to other previously found motion vectors to determine if it has been seen before, and if so contributes its weight to that vector, otherwise enters a new vector in the list. The above and left vectors have twice the weight of the above-left vector.

As is the case with many contexts used by VP8, it is possible for macroblocks near the top or left edges of the image to reference blocks that are outside the visible image. VP8 provides a border of 1 macroblock filled with 0x0 motion vectors left of the left edge, and a border filled with 0,0 motion vectors of 1 macroblocks above the top edge.

Much of the process is more easily described in C than in English. The reference code for this can be found in `findnearmv.c`. The calculation of reference vectors, probability table, and, finally, the inter-prediction mode itself is implemented as follows.

```
typedef union
{
    unsigned int as_int;
    MV           as_mv;
} int_mv;        /* facilitates rapid equality tests */


static void mv_bias(MODE_INFO *x,int refframe, int_mv *mvp, int * ref_frame_sign_bias )
{
    MV xmv;
    xmv = x->mbmi.mv.as_mv;
    if ( ref_frame_sign_bias[x->mbmi.ref_frame] != ref_frame_sign_bias[refframe] )
    {
        xmv.row*=-1;
        xmv.col*=-1;
    }
    mvp->as_mv = xmv;
}
```

```
void vp8_clamp_mv(MV *mv, const MACROBLOCKD *xd)
{
    if ( mv->col < (xd->mb_to_left_edge - LEFT_TOP_MARGIN) )
        mv->col = xd->mb_to_left_edge - LEFT_TOP_MARGIN;
    else if ( mv->col > xd->mb_to_right_edge + RIGHT_BOTTOM_MARGIN )
        mv->col = xd->mb_to_right_edge + RIGHT_BOTTOM_MARGIN;

    if ( mv->row < (xd->mb_to_top_edge - LEFT_TOP_MARGIN) )
        mv->row = xd->mb_to_top_edge - LEFT_TOP_MARGIN;
    else if ( mv->row > xd->mb_to_bottom_edge + RIGHT_BOTTOM_MARGIN )
        mv->row = xd->mb_to_bottom_edge + RIGHT_BOTTOM_MARGIN;
}
```

In the function `vp8_find_near_mvs()`, the vectors “nearest” and “near” are used by the corresponding modes.

The vector `best_mv` is used as a base for explicitly-coded motion vectors.

The first three entries in the return value `cnt` are (in order) weighted census values for “zero”, “nearest”, and “near” vectors. The final value indicates the extent to which `SPLIT_MV` was used by the neighboring macroblocks. The largest possible “weight” value in each case is 5.

```
void vp8_find_near_mvs
(
    MACROBLOCKD *xd,
    const MODE_INFO *here,
    MV *nearest,
    MV *near,
    MV *best_mv,
    int cnt[4],
    int refframe,
    int * ref_frame_sign_bias
)
{
    const MODE_INFO *above = here - xd->mode_info_stride;
    const MODE_INFO *left = here - 1;
    const MODE_INFO *aboveleft = above - 1;
    int_mv            near_mvs[4];
    int_mv           *mv = near_mvs;
    int              *cntx = cnt;
    enum {CNT_ZERO, CNT_NEAREST, CNT_NEAR, CNT_SPLITMV};

    /* Zero accumulators */
    mv[0].as_int = mv[1].as_int = mv[2].as_int = 0;
    cnt[0] = cnt[1] = cnt[2] = cnt[3] = 0;

    /* Process above */
    if(above->mbmi.ref_frame != INTRA_FRAME) {
        if(above->mbmi.mv.as_int) {
            (++mv)->as_int = above->mbmi.mv.as_int;
            mv_bias(above, refframe, mv, ref_frame_sign_bias);
            ++cntx;
        }
        *cntx += 2;
    }

    /* Process left */
    if(left->mbmi.ref_frame != INTRA_FRAME) {
        if(left->mbmi.mv.as_int) {
            int_mv this_mv;

            this_mv.as_int = left->mbmi.mv.as_int;
            mv_bias(left, refframe, &this_mv, ref_frame_sign_bias);

            if(this_mv.as_int != mv->as_int) {
                (++mv)->as_int = this_mv.as_int;
                ++cntx;
            }
```

```
            *cntx += 2;
        } else
            cnt[CNT_ZERO] += 2;
    }

    /* Process above left */
    if(aboveleft->mbmi.ref_frame != INTRA_FRAME) {
        if(aboveleft->mbmi.mv.as_int) {
            int_mv this_mv;

            this_mv.as_int = aboveleft->mbmi.mv.as_int;
            mv_bias(aboveleft, refframe, &this_mv, ref_frame_sign_bias);

            if(this_mv.as_int != mv->as_int) {
                (++mv)->as_int = this_mv.as_int;
                ++cntx;
            }
            *cntx += 1;
        } else
            cnt[CNT_ZERO] += 1;
    }

    /* If we have three distinct MV's ... */
    if(cnt[CNT_SPLITMV]) {
        /* See if above-left MV can be merged with NEAREST */
        if(mv->as_int == near_mvs[CNT_NEAREST].as_int)
            cnt[CNT_NEAREST] += 1;
    }

    cnt[CNT_SPLITMV] = ((above->mbmi.mode == SPLITMV)
                        + (left->mbmi.mode == SPLITMV)) * 2
                       + (aboveleft->mbmi.mode == SPLITMV);

    /* Swap near and nearest if necessary */
    if(cnt[CNT_NEAR] > cnt[CNT_NEAREST]) {
        int tmp;
        tmp = cnt[CNT_NEAREST];
        cnt[CNT_NEAREST] = cnt[CNT_NEAR];
        cnt[CNT_NEAR] = tmp;
        tmp = near_mvs[CNT_NEAREST].as_int;
        near_mvs[CNT_NEAREST].as_int = near_mvs[CNT_NEAR].as_int;
        near_mvs[CNT_NEAR].as_int = tmp;
    }

    /* Use near_mvs[0] to store the "best" MV */
    if(cnt[CNT_NEAREST] >= cnt[CNT_ZERO])
        near_mvs[CNT_ZERO] = near_mvs[CNT_NEAREST];

    /* Set up return values */
    *best_mv = near_mvs[0].as_mv;
    *nearest = near_mvs[CNT_NEAREST].as_mv;
    *near = near_mvs[CNT_NEAR].as_mv;

    vp8_clamp_mv(nearest, xd);
    vp8_clamp_mv(near, xd);
    vp8_clamp_mv(best_mv, xd); //TODO: move this up before the copy
}
```

The `mv_ref` probability table (`mv_ref_p`) is then derived from the census as follows.

```
const int vp8_mode_contexts[6][4] =
{
  {   7,     1,     1,   143,   },
  {  14,    18,    14,   107,   },
  { 135,    64,    57,    68,   },
  {  60,    56,   128,    65,   },
  { 159,   134,   128,    34,   },
  { 234,   188,   128,    28,   },
}
```

```
vp8_prob *vp8_mv_ref_probs(vp8_prob mv_ref_p[VP8_MVREFS-1], int cnt[4])
{
    mv_ref_p[0] = vp8_mode_contexts [cnt[0]] [0];
    mv_ref_p[1] = vp8_mode_contexts [cnt[1]] [1];
    mv_ref_p[2] = vp8_mode_contexts [cnt[2]] [2];
    mv_ref_p[3] = vp8_mode_contexts [cnt[3]] [3];
    return p;
}
```

Once `mv_ref_p` is established, the `mv_ref` is decoded as usual.

```
  mvr = (mv_ref) treed_read( d, mv_ref_tree, mv_ref_p);
```

For the first four inter-coding modes, the same motion vector is used for all the Y subblocks. The first three modes use an implicit motion vector.

Mode	Instruction
`mv_nearest`	Use the nearest vector returned by `vp8_find_near_mvs` .
`mv_near`	Use the near vector returned by `vp8_find_near_mvs` .
`mv_zero`	Use a zero vector, that is, predict the current macroblock from the corresponding macroblock in the prediction frame.
`NEWMV`	This mode is followed by an explicitly-coded motion vector (the format of which is described in the next chapter) that is added (component-wise) to the `best_mv` reference vector returned by `find_near_mvs` and applied to all 16 subblocks.

16.4 Split Prediction

The remaining mode (`SPLITMV`) causes multiple vectors to be applied to the Y subblocks. It is immediately followed by a partition specification that determines how many vectors will be specified and how they will be assigned to the subblocks. The possible partitions, with indicated subdivisions and coding tree, are as follows.

```
typedef enum
{
    mv_top_bottom,    /* two pieces {0...7} and {8...15} */
    mv_left_right,    /* {0,1,4,5,8,9,12,13} and {2,3,6,7,10,11,14,15} */
    mv_quarters,     /* {0,1,4,5}, {2,3,6,7}, {8,9,12,13}, {10,11,14,15} */
    MV_16,           /* every subblock gets its own vector {0} ... {15} */

    mv_num_partitions
}
MVpartition;

const tree_index mvpartition_tree [2 * (mvnum_partition - 1)] =
{
 -MV_16, 2,                          /* MV_16 = "0" */
  -mv_quarters, 4,                   /* mv_quarters = "10" */
   -mv_top_bottom, -mv_left_right     /* top_bottom = "110", left_right = "111" */
};
```

The partition is decoded using a fixed, constant probability table:

```
const Prob mvpartition_probs [mvnum_partition - 1] = { 110, 111, 150};
part = (MVpartition) treed_read( d, mvpartition_tree, mvpartition_probs);
```

After the partition come two (for `mv_top_bottom` or `mv_left_right`), four (for `mv_quarters`), or sixteen (for `MV_16`) subblock inter-prediction modes. These modes occur in the order indicated by the partition layouts (given as comments to the `MVpartition` enum) and are coded as follows. (As was done for the macroblock-level modes, we offset the mode enumeration so that a single variable may unambiguously hold either an intra- or inter-subblock mode.)

```
typedef enum
{
    LEFT4x4 = num_intra_bmodes,   /* use already-coded MV to my left */
    ABOVE4x4,                    /* use already-coded MV above me */
    ZERO4x4,                     /* use zero MV */
    NEW4x4,                      /* explicit offset from "best" */

    num_sub_mv_ref
};
sub_mv_ref;

const tree_index sub_mv_ref_tree [2 * (num_sub_mv_ref - 1)] =
{
 -LEFT4X4, 2,           /* LEFT = "0" */
```

```
  -ABOVE4X4, 4,          /* ABOVE = "10" */
   -ZERO4X4, -NEW4X4     /* ZERO = "110", NEW = "111" */
};

/* Constant probabilities and decoding procedure. */

const Prob sub_mv_ref_prob [num_sub_mv_ref - 1] = { 180, 162, 25};

    sub_ref = (sub_mv_ref) treed_read( d, sub_mv_ref_tree, sub_mv_ref_prob);
```

The first two sub-prediction modes simply copy the already-coded motion vectors used by the blocks above and to-the-left of the subblock at the upper left corner of the current subset (i.e., collection of subblocks being predicted). These prediction blocks need not lie in the current macroblock and, if the current subset lies at the top or left edges of the frame, need not lie in the frame. In this latter case, their motion vectors are taken to be zero, as are subblock motion vectors within an intra-predicted macroblock. Also, to ensure the correctness of prediction within this macroblock, all subblocks lying in an already-decoded subset of the current macroblock must have their motion vectors set.

`ZERO4x4` uses a zero motion vector and predicts the current subset using the corresponding subset from the prediction frame.

NEW4x4 is exactly like NEWMV except applied only to the current subset. It is followed by a 2-dimensional motion vector offset (described in the next chapter) that is added to the best vector returned by the earlier call to find_near_mvs to form the motion vector in effect for the subset.

Parsing of both inter-prediction modes and motion vectors (described next) can be found in the reference decoder file `decodemv.c`.

Chapter 17: Motion Vector Decoding

As discussed above, motion vectors appear in two places in the VP8 datastream: applied to whole macroblocks in `NEWMV` mode and applied to subsets of macroblocks in `NEW4x4` mode. The format of the vectors is identical in both cases.

Each vector has two pieces: A vertical component (row) followed by a horizontal component (column). The row and column use separate coding probabilities but are otherwise represented identically.

17.1 Coding of Each Component

Each component is a signed integer V representing a vertical or horizontal luma displacement of V quarter-pixels (and a chroma displacement of V eighth-pixels). The absolute value of V, if non-zero, is followed by a boolean sign. V may take any value between -255 and +255, inclusive.

The absolute value A is coded in one of two different ways according to its size. For $0 \leq A \leq 7$, A is tree-coded, and for $8 \leq A \leq 255$, the bits in the binary expansion of A are coded using independent boolean probabilities. The coding of A begins with a bool specifying which range is in effect.

Decoding a motion vector component then requires a 17-position probability table, whose offsets, along with the procedure used to decode components, are as follows:

```
typedef enum
{
    mvpis_short,             /* short (<= 7) vs long (>= 8) */
    MVPsign,                /* sign for non-zero */
    MVPshort,               /* 8 short values = 7-position tree */

    MVPbits = MVPshort + 7, /* 8 long value bits w/independent probs */

    MVPcount = MVPbits + 8  /* 17 probabilities in total */
}
MVPindices;

typedef Prob MV_CONTEXT [MVPcount];/* Decoding spec for a single component */


/* Tree used for small absolute values (has expected correspondence). */

const tree_index small_mvtree [2 * (8 - 1)] =
{
 2, 8,          /* "0" subtree, "1" subtree */
  4, 6,         /* "00" subtree", "01" subtree */
   -0, -1,      /* 0 = "000", 1 = "001" */
   -2, -3,      /* 2 = "010", 3 = "011" */
  10, 12,       /* "10" subtree, "11" subtree */
   -4, -5,      /* 4 = "100", 5 = "101" */
   -6, -7       /* 6 = "110", 7 = "111" */
};

/* Read MV component at current decoder position, using supplied probs. */
```

```
int read_mvcomponent( bool_decoder *d, const MV_CONTEXT *mvc)
{
    const Prob * const p = (const Prob *) mvc;
    int A = 0;

    if( read_bool( d, p [mvpis_short]))        /* 8 <= A <= 255 */
    {
        /* Read bits 0, 1, 2 */

        int i = 0;
        do { A += read_bool( d, p [MVPbits + i]) << i;}  while( ++i < 3);

        /* Read bits 7, 6, 5, 4 */

        i = 7;
        do { A += read_bool( d, p [MVPbits + i]) << i;}  while( --i > 3);

        /* We know that A >= 8 because it is coded long,
           so if A <= 15, bit 3 is one and is not explicitly coded. */

        if( !(A & 0xfff0)  ||  read_bool( d, p [MVPbits + 3]))
            A += 8;
    }
    else                                    /* 0 <= A <= 7 */
        A = treed_read( d, small_mvtree, p + MVPshort);

    return A && read_bool( r, p [MVPsign]) ?  -A : A;
}
```

17.2 Probability Updates

The decoder should maintain an array of two `MV_CONTEXT`s for decoding row and column components, respectively. These `MV_CONTEXT`s should be set to their defaults every key frame. Each individual probability may be updated every interframe (by field J of the frame header) using a constant table of update probabilities. Each optional update is of the form $B?P(7)$, that is, a bool followed by a 7-bit probability specification if true.

As with other dynamic probabilities used by VP8, the updates remain in effect until the next key frame or until replaced via another update.

In detail, the probabilities should then be managed as follows.

```
/* Never-changing table of update probabilities for each individual
   probability used in decoding motion vectors. */

const MV_CONTEXT vp8_mv_update_probs[2] =
{
  {
    237,
    246,
    253, 253, 254, 254, 254, 254, 254,
    254, 254, 254, 254, 254, 250, 250, 252, 254, 254
  },
```

```
  {
    231,
    243,
    245, 253, 254, 254, 254, 254, 254,
    254, 254, 254, 254, 254, 251, 251, 254, 254, 254
  }
};

/* Default MV decoding probabilities. */

const MV_CONTEXT default_mv_context[2] =
{
  {                         // row
    162,                      // is short
    128,                      // sign
      225, 146, 172, 147, 214,  39, 156,       // short tree
    128, 129, 132,  75, 145, 178, 206, 239, 254, 254 // long bits
  },

  {                         // same for column
    164,                      // is short
    128,
    204, 170, 119, 235, 140, 230, 228,
    128, 130, 130,  74, 148, 180, 203, 236, 254, 254 // long bits

  }
};

/* Current MV decoding probabilities, set to above defaults every key frame. */

MV_CONTEXT mvc [2];      /* always row, then column */

/* Procedure for decoding a complete motion vector. */

typedef struct { int16 row, col;}  MV;  /* as in previous chapter */

MV read_mv( bool_decoder *d)
{
    MV v;
    v.row = (int16) read_mvcomponent( d, mvc);
    v.col = (int16) read_mvcomponent( d, mvc + 1);
    return v;
}

/* Procedure for updating MV decoding probabilities, called every interframe
   with "d" at the appropriate position in the frame header. */

void update_mvcontexts( bool_decoder *d)
{
    int i = 0;
    do {                                    /* component = row, then column */
        const Prob *up = mv_update_probs[i];  /* update probs for component */
        Prob *p = mvc[i];                   /* start decode tbl "" */
        Prob * const pstop = p + MVPcount;  /* end decode tbl "" */
        do {
            if( read_bool( d, *up++))        /* update this position */
```

```
            {
                const Prob x = read_literal( d, 7);

                *p = x? x<<1 : 1;
            }
        } while( ++p < pstop);              /* next position */
    } while( ++i < 2);                      /* next component */
}
```

This completes the description of the motion-vector decoding procedure and, with it, the procedure for decoding interframe macroblock prediction records.

Chapter 18: Interframe Prediction

Given an inter-prediction specification for the current macroblock, that is, a reference frame together with a motion vector for each of the sixteen Y subblocks, we describe the calculation of the prediction buffer for the macroblock. Frame reconstruction is then completed via the previously-described processes of residue summation (Chapter 14) and loop filtering (Chapter 15).

The management of inter-predicted subblocks may be found in the reference decoder file `reconinter.c`; sub-pixel interpolation is implemented in `filter_c.c`.

18.1 Bounds on and Adjustment of Motion Vectors

It is possible within the VP8 format for a block or macroblock to have an arbitrarily large motion vectors, due to the fact that each motion vector is differentially encoded without any clamp from a neighboring block or macroblock.

Because the motion vectors applied to the chroma subblocks have 1/8 pixel resolution, the synthetic pixel calculation, outlined in Chapter 5 and detailed below, uses this resolution for the luma subblocks as well. In accordance, the stored luma motion vectors are all doubled, each component of each luma vector becoming an even integer in the range -510 to +510, inclusive.

The vector applied to each chroma subblock is calculated by averaging the vectors for the 4 luma subblocks occupying the same visible area as the chroma subblock in the usual correspondence, that is, the vector for U and V block 0 is the average of the vectors for the Y subblocks { 0, 1, 4, 5}, chroma block 1 corresponds to Y blocks { 2, 3, 6, 7}, chroma block 2 to Y blocks { 8, 9, 12, 13}, and chroma block 3 to Y blocks { 10, 11, 14, 15}.

In detail, each of the two components of the vectors for each of the chroma subblocks is calculated from the corresponding luma vector components as follows:

```
int avg( int c1, int c2, int c3, int c4)
{
    int s = c1 + c2 + c3 + c4;

    /* The shift divides by 8 (not 4) because chroma pixels have half
       the diameter of luma pixels.  The handling of negative motion vector
       components is slightly cumbersome because, strictly speaking,
       right shifts of negative numbers are not well-defined in C. */

    return s >= 0 ?  (s + 4) >> 3 : -( (-s + 4) >> 3);
}
```

18.2 Prediction Subblocks

The prediction calculation for each subblock is then as follows. Temporarily disregarding the fractional part of the motion vector (that is, rounding “up” or “left” by right-shifting each component 3 bits with sign propagation) and adding the origin (upper left position) of the (16x16 luma or 8x8 chroma) current macroblock gives us an origin in the Y, U, or V plane of the predictor frame (either the golden frame or previous frame).

Considering that origin to be the upper left corner of a (luma or chroma) macroblock, we need to specify the relative positions of the pixels associated to that subblock, that is, any pixels that might be involved in the sub-pixel interpolation processes for the subblock.

18.3 Sub-pixel Interpolation

The sub-pixel interpolation is effected via two one-dimensional convolutions. These convolutions may be thought of as operating on a two-dimensional array of pixels whose origin is the subblock origin, that is the origin of the prediction macroblock described above plus the offset to the subblock. Because motion vectors are arbitrary, so are these “prediction subblock origins”.

The integer part of the motion vector is subsumed in the origin of the prediction subblock, the 16 (synthetic) pixels we need to construct are given by 16 offsets from the origin. The integer part of each of these offsets is the offset of the corresponding pixel from the subblock origin (using the vertical stride). To these integer parts is added a constant fractional part, which is simply the difference between the actual motion vector and its integer truncation used to calculate the origins of the prediction macroblock and subblock. Each component of this fractional part is an integer between 0 and 7, representing a forward displacement in eighths of a pixel.

It is these fractional displacements that determine the filtering process. If they both happen to be zero (that is, we had a “whole pixel” motion vector), the prediction subblock is simply copied into the corresponding piece of the current macroblock’s prediction buffer. As discussed in Chapter 14, the layout of the macroblock’s prediction buffer can depend on the specifics of the reconstruction implementation chosen. Of course, the vertical displacement between lines of the prediction subblock is given by the stride, as are all vertical displacements used here.

Otherwise, at least one of the fractional displacements is non-zero. We then synthesize the missing pixels via a horizontal, followed by a vertical, one-dimensional interpolation.

The two interpolations are essentially identical. Each uses an (at most) six-tap filter (the choice of which of course depends on the one-dimensional offset). Thus, every calculated pixel references at most three pixels before (above or to-the-left of) it and at most three pixels after (below or to-the-right of) it. The horizontal interpolation must calculate two extra rows above and three extra rows below the 4x4 block, to provide enough samples for the vertical interpolation to proceed.

The exact implementation of subsampling is as follows.

```
/* Filter taps taken to 7-bit precision.
   Because DC is always passed, taps always sum to 128. */

const int filters [8] [6[ = {           /* indexed by displacement */
    { 0,  0,  128,    0,   0,  0 },     /* degenerate whole-pixel */
    { 0, -6,  123,   12,  -1,  0 },     /* 1/8 */
    { 2, -11, 108,   36,  -8,  1 },     /* 1/4 */
    { 0, -9,   93,   50,  -6,  0 },     /* 3/8 */
    { 3, -16,  77,   77, -16,  3 },     /* 1/2 is symmetric */
    { 0, -6,   50,   93,  -9,  0 },     /* 5/8 = reverse of 3/8 */
    { 1, -8,   36,  108, -11,  2 },     /* 3/4 = reverse of 1/4 */
    { 0, -1,   12,  123,  -6,  0 }      /* 7/8 = reverse of 1/8 */
};
```

```
/* One-dimensional synthesis of a single sample.
   Filter is determined by fractional displacement */

Pixel interp(
    const int fil[6],   /* filter to apply */
    const Pixel *p,     /* origin (rounded "before") in prediction area */
    const int s         /* size of one forward step "" */
) {
    int32 a = 0;
    int i = 0;
    p -= s + s;         /* move back two positions */

    do {
        a += *p * fil[i];
        p += s;
    }  while( ++i < 6);

    return clamp255( (a + 64) >> 7);    /* round to nearest 8-bit value */
}


/* First do horizontal interpolation, producing intermediate buffer. */

void Hinterp(
    Pixel temp[9][4],   /* 9 rows of 4 (intermediate) destination values */
    const Pixel *p,     /* subblock origin in prediction frame */
    int s,              /* vertical stride to be used in prediction frame */
    uint hfrac          /* 0 <= horizontal displacement <= 7 */
) {
    const int * const fil = filters [hfrac];

    int r = 0;  do              /* for each row */
    {
        int c = 0;  do          /* for each destination sample */
        {
            /* Pixel separation = one horizontal step = 1 */

            temp[r][c] = interp( fil, p + c, 1);
        }
        while( ++c < 4);
    }
    while( p += s, ++r < 9);    /* advance p to next row */
}

/* Finish with vertical interpolation, producing final results.
   Input array "temp" is of course that computed above. */


void Vinterp(
    Pixel final[4][4],      /* 4 rows of 4 (final) destination values */
    const Pixel temp[9][4],
    uint vfrac              /* 0 <= vertical displacement <= 7 */
) {
    const int * const fil = filters [vfrac];

    int r = 0;  do              /* for each row */
```

```
    {
        int c = 0;  do          /* for each destination sample */
        {
            /* Pixel separation = one vertical step = width of array = 4 */

            final[r][c] = interp( fil, temp[r] + c, 4);
        }
        while( ++c < 4);
    }
    while( ++r < 4);
}
```

18.4 Filter Properties

We discuss briefly the rationale behind the choice of filters. Our approach is necessarily cursory; a genuinely accurate discussion would require a couple of books. Readers unfamiliar with signal processing may or may not wish to skip this.

All digital signals are of course sampled in some fashion. The case where the inter-sample spacing (say in time for audio samples, or space for pixels) is uniform, that is, the same at all positions, is particularly common and amenable to analysis. Many aspects of the treatment of such signals are best-understood in the frequency domain via Fourier Analysis, particularly those aspects of the signal that are not changed by shifts in position, especially when those positional shifts are not given by a whole number of samples.

Non-integral translates of a sampled signal are a textbook example of the foregoing. In our case of non-integral motion vectors, we wish to say what the underlying image “really is” at these pixels we don’t have values for but feel that it makes sense to talk about. The correctness of this feeling is predicated on the underlying signal being band-limited, that is, not containing any energy in spatial frequencies that cannot be faithfully rendered at the pixel resolution at our disposal. In one dimension, this range of “OK” frequencies is called the Nyquist band; in our two-dimensional case of integer-grid samples, this range might be termed a Nyquist rectangle. The finer the grid, the more we know about the image, and the wider the Nyquist rectangle.

It turns out that, for such band-limited signals, there is indeed an exact mathematical formula to produce the correct sample value at an arbitrary point. Unfortunately, this calculation requires the consideration of every single sample in the image, as well as needing to operate at infinite precision. Also, strictly speaking, all band-limited signals have infinite spatial (or temporal) extent, so everything we are discussing is really some sort of approximation.

It is true that the theoretically correct subsampling procedure, as well as any approximation thereof, is always given by a translation-invariant weighted sum (or filter) similar to that used by VP8. It is also true that the reconstruction error made by such a filter can be simply represented as a multiplier in the frequency domain, that is, such filters simply multiply the Fourier transform of any signal to which they are applied by a fixed function associated to the filter. This fixed function is usually called the frequency response (or transfer function); the ideal subsampling filter has a frequency response equal to one in the Nyquist rectangle and zero everywhere else.

Another basic fact about approximations to “truly correct” subsampling is that, the wider the subrectangle (within the Nyquist rectangle) of spatial frequencies one wishes to “pass” (that is, correctly render) or, put more accurately, the closer one wishes to approximate the ideal transfer function, the more samples of the original signal must be considered by the subsampling, and the wider the calculation precision necessitated.

The filters chosen by VP8 were chosen, within the constraints of 4 or 6 taps and 7-bit precision, to do the best possible job of handling the low spatial frequencies near the zeroth DC frequency along with introducing no resonances (places where the absolute value of the frequency response exceeds one).

The justification for the foregoing has two parts. First, resonances can produce extremely objectionable visible artifacts when, as often happens in actual compressed video streams, filters are applied repeatedly. Second, the vast majority of energy in real-world images lies near DC and not at the high-end; also, roughly speaking, human perception tends to be more sensitive at these lower spatial frequencies.

To get slightly more specific, the filters chosen by VP8 are the best resonance-free 4- or 6-tap filters possible, where “best” describes the frequency response near the origin: the response at 0 is required to be 1 and the graph of the response at 0 is as flat as possible.

To provide an intuitively more obvious point of reference, the “best” 2-tap filter is given by simple linear interpolation between the surrounding actual pixels.

Finally, it should be noted that, because of the way motion vectors are calculated, the (shorter) 4-tap filters (used for odd fractional displacements) are applied in the chroma plane only. Human color perception is notoriously poor, especially where higher spatial frequencies are involved. The shorter filters are easier to understand mathematically, and the difference between them and a theoretically slightly better 6-tap filter is negligible where chroma is concerned.

Chapter 19: References

1. “ITU BT.601: Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios”, International Telecommunication Union, Jan, 2007
2. *The C Programming Language (2nd edition)* by Brian Kernighan and Dennis Ritchie, Prentice Hall, Englewood Cliffs, NJ, USA, 1988
3. “A Mathematical Theory of Communication” by C.E. Shannon, *Bell System Technical Journal*, vol. 27, pp. 379-423, 623-656, July, October, 1948

Revision History

Date	By	Note
05/14/2010	Wilkins, Xu, Quillio	Edits and updates related to WebM Project launch.
04/22/2010	Paul Wilkins	Updates and corrections.
05/03/2009	Lou Quillio	Edits for clarity; refactoring; style imposition; reformatting.
3/30/2009	Suman Sunkara	Updated with editorial changes from review.
3/22/2009	Yaowu Xu	Updated with new bitstream change.
1/20/2009	Yaowu Xu	Incorporated changes from “VP7 Data Format and Decoding Specification” by Tim Murphy, Jim Bankoski, et al.
1/12/2009	Yaowu Xu	Created document based on “VP7 Data Format and Decoder Overview” by Tim Murphy.