

Network Working Group
Internet-Draft
Intended status: Informational
Expires: November 19, 2011

J. Bankoski
P. Wilkins
Y. Xu
Google, Inc.
May 18, 2011

VP8 Data Format and Decoding Guide
draft-bankoski-vp8-bitstream-02

Abstract

This document describes the VP8 compressed video data format, together with a discussion of the decoding procedure for the format.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 19, 2011.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1.	Introduction	4
2.	Format Overview	6
3.	Compressed Frame Types	8
4.	Overview of Compressed Data Format	9
5.	Overview of the Decoding Process	11
6.	Description of Algorithms	16
7.	Boolean Entropy Decoder	19
7.1.	Underlying Theory of Coding	20
7.2.	Practical Algorithm Description	21
7.3.	Actual Implementation	23
8.	Compressed Data Components	28
8.1.	Tree Coding Implementation	30
8.2.	Tree Coding Example	31
9.	Frame Header	34
9.1.	Uncompressed Data Chunk	34
9.2.	Color Space and Pixel Type (Key Frames-only)	37
9.3.	Segment-based Adjustments	37
9.4.	Loop Filter Type and Levels	38
9.5.	Token Partition and Partition Data Offsets	39
9.6.	Dequantization Indices	40
9.7.	Refresh Golden Frame and AltRef Frame	41
9.8.	Refresh Last Frame Buffer	42
9.9.	DCT Coefficient Probability Update	42
9.10.	Remaining Frame Header Data (non-Key Frame)	43
9.11.	Remaining Frame Header Data (Key Frame)	44
10.	Segment-based Feature Adjustments	45
11.	Key Frame Macroblock Prediction Records	46
11.1.	mb_skip_coeff	46
11.2.	Luma Modes	46
11.3.	Subblock Mode Contexts	49
11.4.	Chroma Modes	50
11.5.	Subblock Mode Probability Table	51
12.	Intraframe Prediction	55
12.1.	mb_skip_coeff	55
12.2.	Chroma Prediction	56
12.3.	Luma Prediction	58
13.	DCT Coefficient Decoding	65
13.1.	MB Without non-Zero Coefficient Values	65
13.2.	Coding of Individual Coefficient Values	66
13.3.	Token Probabilities	68
13.4.	Token Probability Updates	72

13.5.	Default Token Probability Table	77
14.	DCT and WHT Inversion and Macroblock Reconstruction	82
14.1.	Dequantization	82
14.2.	Inverse Transforms	83
14.3.	Implementation of the WHT Inversion	84

14.4.	Implementation of the DCT Inversion	86
14.5.	Summation of Predictor and Residue	89
15.	Loop Filter	90
15.1.	Filter Geometry and Overall Procedure	91
15.2.	Simple Filter	93
15.3.	Normal Filter	97
15.4.	Calculation of Control Parameters	102
16.	Interframe Macroblock Prediction Records	104
16.1.	Intra-Predicted Macroblocks	104
16.2.	Inter-Predicted Macroblocks	105
16.3.	Mode and Motion Vector Contexts	106
16.4.	Split Prediction	112
17.	Motion Vector Decoding	116
17.1.	Coding of Each Component	116
17.2.	Probability Updates	118
18.	Interframe Prediction	121
18.1.	Bounds on and Adjustment of Motion Vectors	121
18.2.	Prediction Subblocks	122
18.3.	Sub-pixel Interpolation	123
18.4.	Filter Properties	126
19.	Annex A: Bitstream Syntax	129
19.1.	Uncompressed Data Chunk	129
19.2.	Frame Header	131
19.3.	Macroblock Data	143
20.	Attachment One: Reference Decoder Source Code	147
20.1.	bit_ops.h	147
20.2.	bool_decoder.h	147
20.3.	dequant_data.h	151
20.4.	dixie.c	151
20.5.	dixie.h	163
20.6.	dixie_loopfilter.c	170
20.7.	dixie_loopfilter.h	182
20.8.	idct_add.c	182
20.9.	idct_add.h	186
20.10.	mem.h	186
20.11.	modemv.c	188

20.12.	modemv.h	203
20.13.	modemv_data.h	203
20.14.	predict.c	208
20.15.	predict.h	238
20.16.	tokens.c	238
20.17.	tokens.h	248
20.18.	vp8_prob_data.h	257
20.19.	vpx_codec_internal.h	266
20.20.	vpx_decoder.h	276
20.21.	vpx_integer.h	283
21.	References	286
	Authors' Addresses	287

[1.](#) Introduction

This document describes the VP8 compressed video data format, together with a discussion of the decoding procedure for the format. It is intended to be used in conjunction with and as a guide to the reference decoder source code provided in Attachment One. If there are any conflicts between this narrative 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 narrative.

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" transform (hereafter abbreviated as WHT) 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.

[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 in the VP8 bitstream header for the support of a secondary YUV color format, in the form of a reserved bit.

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 than 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`.

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.

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.

[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 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

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 [[Kernighan](#)].

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.

----- Begin code block -----

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

typedef short int16; /* signed int 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;

----- End code block -----
```

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 `pi`; 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.

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 booleans 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, [\[Shannon\]](#) proved that there is a lower bound for the average data rate of a faithful encoding of a sequence of booleans (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

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

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 [\[Bell\]](#).

[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 $0 \leq x < 1$. 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 $\text{pow}(2, -j) * b[j]$ or $\text{pow}(256, -j) * B[j]$.

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 p_1 is the

probability of the first bool being zero and a zero is coded, the range of possible x is restricted to $0 \leq x < p_1$. If a one is coded, the range becomes $p_1 \leq x < 1$.

The coding continues by repeating the same idea. At every stage, there is an interval $a \leq x < b$ 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 $a \leq x < a + (p(b-a))$. If a one is coded, the possible x are restricted to $a + (p(b-a)) \leq x < b$.

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 $0 \leq x < 1$. To decode the first bool, the decoder is given the first probability p_1 . If $x < p_1$, a zero is decoded; if $x \geq p_1$, 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 $a \leq x < b$ is the current interval and we are to decode a bool with zero-probability p , we return a zero if $a \leq x < a + (p(b-a))$ and a one if $a + (p(b-a)) \leq x < b$. 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 $0 \leq p \leq 255$; 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 $\text{pow}(2, -n)$ 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 $S = \text{pow}(2, -n - 8)$ 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 $v = x$):

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

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 $\text{prob} / 256$ -- and whose probability of being one is $(256 - \text{prob}) / 256$ -- 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 + (((range - 1) * probability))] >> 8)
```

split is approximately $(\text{prob} / 256) * \text{range}$ 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 the incoming boolean *val* to be encoded is false, we leave the left interval endpoint *bottom* alone and reduce *range*, replacing it by *split*. If the incoming *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 *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 *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 $\text{value} < \text{split}$, the bool is zero, and range is replaced with split. If $\text{value} \geq \text{split}$, the bool is one, range is replaced with $\text{range} - \text{split}$, and value is replaced with $\text{value} - \text{split}$.

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 $\text{Value} < \text{Range}$ 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`.

----- Begin code block -----

```
/* 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 8 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 2 input bytes */
            while( ++i <= 2)

```

```

        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 1/28 (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;
}

```

---- End code block -----

[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/256 of being 0. 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 $x \gg 1 : 1$.
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.

```
----- Begin code block -----

/* 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 */
}

----- End code block -----
```

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.

```
----- Begin code block -----
```

```

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);

```

---- End code block -----

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.

[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.

Version number enables or disables certain features in the bitstream, as follows:

Version	Reconstruction filter	Loop filter
0	Bicubic	Normal
1	Bilinear	Simple
2	Bilinear	None
3	None	None
Other	Reserved for future use	

The reference software also adjusts the loop filter based on version number, as per the table above. Version number 1 implies a "simple" loop filter and version numbers 2 and 3 imply no loop filter. However, the "simple" filter setting in this context has no effect whatsoever on the decoding process, and the "no loop filter" setting

only forces the reference encoder to set filter level equal to 0. Neither affect the decoding process. In decoding, the only loop filter settings that matter are those in the frame header.

For key frames the frame tag is followed by a further 7 bytes of uncompressed data, as follows:

---- Begin code block -----

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)

---- End code block -----

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

---- Begin code block -----

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

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

---- End code block -----

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

The following code reads the image dimension from the bitstream:

---- Begin code block -----

```
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;
```

---- End code block -----

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

---- Begin code block -----

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


#endif

---- End code block -----

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 $ew = 16 - (\text{width} \& 15)$ and $eh = 16 - (\text{height} \& 15)$ 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.

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:

- o 0 - YUV color space similar to the YCrCb color space defined in [\[ITU-R BT.601\]](#)
- o 1 - Reserved for future use

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

- o 0 - Decoders are required to clamp the reconstructed pixel values to between 0 and 255 (inclusive).
- o 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 at macroblock level is provided by a subsection in the frame header, which 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	
+-----+	+-----+	+-----+	+-----+

Offsets are embedded in the bitstream to provide the decoder direct access to token partitions. If the number of data partitions is greater than 1, the size of each partition (except the last) is

written in 3 bytes (24 bits). The size of the last partition is the remainder of the data not used by any of the previous partitions. The partitioned data are consecutive in the bitstream, so the size can also be used to calculate the offset of each partition. The following pseudo code illustrates how the size/offset is defined by the three bytes in the bitstream.

---- Begin code block -----

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

---- End code block -----

[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:

---- Begin code block -----

```

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 */

---- End code block -----

```

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 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:

- o 0 means no buffer is copied to golden frame
- o 1 means last_frame is copied to golden frame
- o 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:

- o 0 means no buffer is copied to altref frame
- o 1 means last_frame is copied to altref frame
- o 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.

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 ...

---- Begin code block -----

```
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 */
}
```

---- End code block -----

... 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`.

[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:

---- Begin code block -----

```
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" */  
};
```

---- End code block -----

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

---- Begin code block -----

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

---- End code block -----

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:

----- Begin code block -----

```
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 = "11100" */
    -B_RD_PRED, -B_VR_PRED,       /* B_RD_PRED = "111010",
                                   B_VR_PRED = "111011" */
    -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" */
};

---- End code block -----

```

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:

---- Begin code block -----

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

---- End code block -----

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 jth luma subblock is then


```

---- Begin code block -----
Bmode = (intra_bmode) treed_read( d, bmode_tree, kf_bmode_prob
    [A] [L]);
---- End code block -----

```

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:

```

---- Begin code block -----
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" */
};
---- End code block -----

```

As for the Y modes (in a key frame), the chroma modes are coded using

a fixed, contextless probability table:

```

---- Begin code block -----
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);
```

---- End code block -----

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.

---- Begin code block -----

```
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},

```

Internet-Draft

VP8 Data Format and Decoding Guide

May 2011

```

    { 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}
  }
};
```

---- End code block -----

[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:

---- Begin code block -----

```
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;
```

---- End code block -----

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.

The following figure gives an example of how TM_PRED works:

----- Begin code block -----

P	A0	A1	A2	A3	A4	A5	A6	A7
L0	X00	X01	X02	X03	X04	X05	X06	X07
L1	X10	X11	X12	X13	X14	X15	X16	X17
L2	X20	X21	X22	X23	X24	X25	X26	X27
L3	X30	X31	X32	X33	X34	X35	X36	X37
L4	X40	X41	X42	X43	X44	X45	X46	X47
L5	X50	X51	X52	X53	X54	X55	X56	X57
L6	X60	X61	X62	X63	X64	X65	X66	X67
L7	X70	X71	X72	X73	X74	X75	X76	X77

----- End code block -----

Where P, As and Ls represent reconstructed pixel values from previously coded blocks, and X00 through X77 represent predicted values for the current block. TM_PRED uses the following equation to calculate X_{ij}:

$$X_{ij} = L_i + A_j - P \quad (i, j=0, 1, 2, 3)$$

The exact algorithm is as follows:

----- Begin code block -----

```
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);
}

```

---- End code block -----

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). For the right-most macroblock in each macroblock row except the top row, the extra pixels shall use the same value as the pixel at position (-1, 15), which is the right-most visible pixel on the line immediately above the macroblock row. For the top macroblock row, all the extra pixels assume a value of 127.

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

```
---- Begin code block -----
```

```
/* 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 + 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]);}
```

```
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:
```

```
        {
```

```
            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] = avg2p( 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];
}
}

---- End code block -----

```

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

[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 0th 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.

----- Begin code block -----

```
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" */
};
```

----- End code block -----

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.

---- Begin code block -----

```
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};
```

---- End code block -----

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:

---- Begin code block -----

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

---- End code block -----

[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.

---- Begin code block -----

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

---- End code block -----

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

- o 0 - Y beginning at coefficient 1 (i.e., Y after Y2)
- o 1 - Y2
- o 2 - U or V
- o 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

---- Begin code block -----

```
coef_bands [c]
```

---- End code block -----

where

---- Begin code block -----

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

---- End code block -----

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.

The below pseudo-code illustrates the decoding process. Note that criteria, functions, etc. delimited with ** are either dependent on decoder architecture or are elaborated on elsewhere in this document.

---- Begin code block -----

```
int block[16] = { 0 }; /* current 4x4 block coeffs */
int firstCoeff = 0;
int plane;
```

```
int ctx2;
int ctx3 = 0; /* the 3rd context referred to in above description */
Prob *probTable;
int token;
int sign;
int absValue;
int extraBits;
bool prevCoeffWasZero = false;
bool currentBlockHasCoeffs = false;
/* base coeff abs values per each category, elem #0 is
   DCT_VAL_CATEGORY1, * #1 is DCT_VAL_CATEGORY2 etc */
int categoryBase[6] = { 5, 7, 11, 19, 35, 67 };

/* Determine plane to use */
if( **current_block_is_Y2_block** )      plane = 0;
else if ( **current_block_is_chroma** )  plane = 2;
else if ( **current_macroblock_has_Y2** ) plane = 1;
else                                     plane = 3;
```



```

/* For luma blocks of an "Y2 macroblock" we skip coeff index #0 */
if( plane == 1 )
    firstCoeff++;

/* Determine whether neighbour 4x4 blocks have coefficients.
   This is dependant of the plane we are currently decoding;
   i.e. we check only coefficients from same plane as current
   block. */
if( **left_neighbor_block_has_coefficients(plane)** )
    ctx3++;
if( **above_neighbor_block_has_coefficients(plane)** )
    ctx3++;

for( i = firstCoeff ; i < 16 ; ++i )
{
    ctx2 = coef_bands[i];
    probTable = coef_probs[plane][ctx2][ctx3];

    /* skip first code (dct_eob) if previous token was DCT_0 */
    if( prevCoeffWasZero )
        token = treed_read ( d, **coef_tree_without_eob**,
                             probTable );
    else
        token = treed_read ( d, coef_tree, probTable );

    if( token == dct_eob )
        break;

    if( token != DCT_0 )

```

```

{
    currentBlockHasCoeffs = true;
if( **token_has_extra_bits(token)** )
{
    extraBits = DCTextra( token );
    absValue =
        categoryBase[**token_to_cat_index(token)**] +
        extraBits;
}
else
{

```

```

        absValue = **token_to_abs_value(token)**;
    }

    sign = read_bool(d, 128);
    block[i] = sign ? -absValue : absValue;
}
else
{
    absValue = 0;
}

/* Set contexts and stuff for next coeff */
if( absValue == 0 )      ctx3 = 0;
else if ( absValue == 1 ) ctx3 = 1;
else                    ctx3 = 2;
prevCoeffWasZero = true;
}

/* Store current block status to decoder internals */
**block_has_coefficients[currentMb][currentBlock]** =
    currentBlockHasCoeffs;

---- End code block -----

```

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:

---- Begin code block -----

```
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);
```

---- End code block -----

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

---- Begin code block -----

```
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}
    }
}
};

```

----- End code block -----

[13.5.](#) Default Token Probability Table

The default token probabilities are as follows.

----- Begin code block -----

```

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}
}
}
}

```

```
};
```

```
---- End code block -----
```

[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.

----- Begin code block -----

```
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,
};
```

----- End code block -----

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 0th 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 0th 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:

----- Begin code block -----

```
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;
}
}

```

----- End code block -----

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

----- Begin code block -----

```

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;
}
}

```

---- End code block -----

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 `idctlm.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 [[Loeffler](#)]. 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:

$$\sqrt{2} * \cos(\pi/8)$$
$$\sqrt{2} * \sin(\pi/8)$$

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

$$x * a = x + x*(a-1)$$

therefore

$$x * \sqrt{2} * \cos(\pi/8) = x + x * (\sqrt{2} * \cos(\pi/8) - 1)$$

----- Begin code block -----

```
/* 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;
}
}

```

----- End code block -----

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

[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 efficiency (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 both of the following two conditions:

1. Macroblock coding mode is neither B_PRED nor SPLTMV; and

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.

Note that the simple filter only applies to luma edges. Chroma edges are left unfiltered.

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:

---- Begin code block -----

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

---- End code block -----

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

---- Begin code block -----

```
/* 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);}
```

---- End code block -----

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 $-255 \leq v \leq +255$.

---- Begin code block -----

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

---- End code block -----

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](#).

---- Begin code block -----

```
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 $p_1 - 3p_0 + 3q_0 - q_1$, which can be thought of as
a refinement of $2(q_0 - p_0)$ and the adjustment is
something like $(q_0 - p_0)/4$. */

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

```
/* 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 $-16 \leq a \leq 15$ at this point. */

```
*P0 = s2u( p0 + b);
```

```
return a;
```

```
}
```

```
---- End code block -----
```

---- Begin code block -----

```
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 */
}
```

---- End code block -----

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.

----- Begin code block -----

```
/* 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;
}

```

---- End code block -----

---- Begin code block -----

```

/* 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;
}

```

---- End code block -----

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 $\frac{3}{8}$ the edge difference, the inner adjustment is approximately $\frac{3}{16}$ the edge difference.

----- Begin code block -----

```
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);
    }
}
}

```

---- End code block -----

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.

---- Begin code block -----

```

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 */
    }
}

```

---- End code block -----

[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.

```
---- Begin code block -----

/* 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;

---- End code block -----
```

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.

```
---- Begin code block -----

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;

---- End code block -----
```

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

---- Begin code block -----

```
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;
}
```

---- End code block -----

[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.

---- Begin code block -----

```
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" */
};
```

---- End code block -----

The probability table used to decode this tree is variable. As described in [Section 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

---- Begin code block -----

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

---- End code block -----

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

---- Begin code block -----

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

---- End code block -----

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.

```
---- Begin code block -----

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" */
};

---- End code block -----
```


[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.

---- Begin code block -----

```
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;
}

```

----- End code block -----

----- Begin code block -----

```

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;
    }

```

---- End code block -----

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.

---- Begin code block -----

```

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
}

```

---- End code block -----

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

---- Begin code block -----

```
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, },
}
```

---- End code block -----

---- Begin code block -----

```
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;
}
```

---- End code block -----

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

---- Begin code block -----

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

---- End code block -----

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.

----- Begin code block -----

```
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" */
};
```

----- End code block -----

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

---- Begin code block -----

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

---- End code block -----

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.)

Prior to decoding each subblock, a decoding tree context is chosen as illustrated in the code snippet below. The context is based on the immediate left and above subblock neighbors, and whether they are

equal, are zero, or a combination of those.

---- Begin code block -----

```
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" */
}
```

```

};

/* Choose correct decoding tree context
 * Function parameters are left subblock neighbor MV and above
 * subblock neighbor MV */
int vp8_mvCont(MV *l, MV*a)
{
    int lez = (l->row == 0 && l->col == 0);    /* left neighbour
                                                is zero */
    int aez = (a->row == 0 && a->col == 0);    /* above neighbour
                                                is zero */
    int lea = (l->row == a->row && l->col == a->col); /* left
                                                neighbour equals above neighbour */

    if(lea && lez)
        return SUBMVREF_LEFT_ABOVE_ZED; /* =4 */

    if(lea)
        return SUBMVREF_LEFT_ABOVE_SAME; /* =3 */

    if(aez)
        return SUBMVREF_ABOVE_ZED; /* =2 */

    if(lez)
        return SUBMVREF_LEFT_ZED; /* =1*/

    return SUBMVREF_NORMAL; /* =0 */
}

```

```

}

/* Constant probabilities and decoding procedure. */

const Prob sub_mv_ref_prob [5][num_sub_mv_ref - 1] = {
    { 147,136,18 },
    { 106,145,1 },
    { 179,121,1 },
    { 223,1 ,34 },
    { 208,1 ,1 } };

sub_ref = (sub_mv_ref) treed_read( d, sub_mv_ref_tree,
    sub_mv_ref_prob[context]);

```

---- End code block -----

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`.

[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 -1023 and +1023, 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 $8 \leq A \leq 1023$, 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 19-position probability table, whose offsets, along with the procedure used to decode components, are as follows:

---- Begin code block -----

```
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 + 10      /* 19 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 <= 1023 */
    {
        /* Read bits 0, 1, 2 */

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

        /* Read bits 9, 8, 7, 6, 5, 4 */

        i = 9;
        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;
}

```

---- End code block -----

[17.2.](#) Probability Updates

The decoder should maintain an array of two MV_CONTEXTs for decoding row and column components, respectively. These MV_CONTEXTs 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 $B \ll P(7)$, 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.

---- Begin code block -----

```
/* 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 */
}

```

---- End code block -----

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

[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 ([Section 14](#)) and loop filtering ([Section 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

Since each motion vector is differentially encoded from a neighboring block or macroblock and the only clamp is to ensure that the referenced motion vector represents a valid location inside a reference frame buffer, it is technically possible within the VP8 format for a block or macroblock to have arbitrarily large motion vectors, up to the size of the input image plus the extended border areas. For practical reasons, VP8 imposes a motion vector size range

limit of [-4096, 4095] full pixels, regardless of image size (VP8 defines 14 raw bits for width and height; 16383x16383 is the maximum possible image size). Bitstream-compliant encoders and decoders shall enforce this limit.

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 -2046 to +2046, 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:

---- Begin code block -----

```
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 twice 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);
}
```

---- End code block -----

Furthermore, if the version number in the frame tag specifies only full-pel chroma motion vectors, then the fractional parts of both components of the vector are truncated to zero, as illustrated in the following pseudo-code (assuming 3 bits of fraction for both luma and chroma vectors):

---- Begin code block -----

```
x = x & (~7);  
y = y & (~7);
```

---- End code block -----

Earlier in this document we described the `vp8_clamp_mv()` function to limit "nearest" and "near" motion vector predictors inside specified margins within the frame boundaries. Additional clamping is performed for `NEW_MV` macroblocks, for which the final motion vector is clamped again after combining the "best" predictor and the differential vector decoded from the stream.

However, the secondary clamping is not performed for `SPLIT_MV` macroblocks, meaning that any subblock's motion vector within the `SPLIT_MV` macroblock may point outside the clamping zone. These non-clamped vectors are also used when determining the decoding tree context for subsequent subblocks' modes in the `vp8_mvCont()` function.

[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.

Depending on the reconstruction filter type given in the field Version Number in the frame tag, either a bicubic or a bilinear tap set is used.

The exact implementation of subsampling is as follows.

---- Begin code block -----

```

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

const int BilinearFilters[8][6] =
{
    { 0, 0, 128, 0, 0, 0 },
    { 0, 0, 112, 16, 0, 0 },
    { 0, 0, 96, 32, 0, 0 },
    { 0, 0, 80, 48, 0, 0 },
    { 0, 0, 64, 64, 0, 0 },
    { 0, 0, 48, 80, 0, 0 },
    { 0, 0, 32, 96, 0, 0 },
    { 0, 0, 16, 112, 0, 0 }
};

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 */
    uint bicubic          /* 1=bicubic filter, 0=bilinear */
) {
    const int * const fil = bicubic ? filters [hfrac] :
        BilinearFilters[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 */
    uint bicubic          /* 1=bicubic filter, 0=bilinear */
) {
    const int * const fil = bicubic ? filters [vfrac] :
        BilinearFilters[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);
}
```

---- End code block -----

[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.

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.

[19.](#) Annex A: Bitstream Syntax

This annex presents the bitstream syntax in a tabular form. All the information elements have been introduced and explained in the previous chapters but are collected here for a quick reference. Each syntax element is shortly described after the tabular representation along with a reference to the corresponding paragraph in the main document. The meaning of each syntax element value is not repeated here.

The top-level hierarchy of the bitstream is introduced in [Section 4](#).

Definition of syntax element coding types can be found in [Section 8](#). The types used in the representation in this annex are:

- o $f(n)$, n -bit value from stream (n successive bits, not boolean encoded)

- o $L(n)$, n -bit number encoded as n booleans (with equal probability of being 0 or 1)
- o $B(p)$, bool with probability p of being 0
- o T , tree-encoded value

19.1. Uncompressed Data Chunk

Frame Tag	Type
frame_tag	f(24)
if (key_frame) {	
start_code	f(24)
horizontal_size_code	f(16)
vertical_size_code	f(16)
}	

The 3-byte frame tag can be parsed as follows:

---- Begin code block -----

```

unsigned char *c = pbi->Source;
unsigned int tmp;

tmp = (c[2] << 16) | (c[1] << 8) | c[0];

key_frame = tmp & 0x1;
version = (tmp >> 1) & 0x7;
show_frame = (tmp >> 4) & 0x1;
first_part_size = (tmp >> 5) & 0x7FFFF;

```

---- End code block -----

Where:

- o key_frame indicates if the current frame is a key frame or not.
- o version determines the bitstream version.
- o show_frame indicates if the current frame is meant to be displayed or not.
- o first_part_size determines the size of the first partition (control partition), excluding the uncompressed data chunk.

The start_code is a constant 3-byte pattern having value 0x9d012a. The latter part of the uncompressed chunk (after the start_code) can be parsed as follows:

---- Begin code block -----

```
unsigned char *c = pbi->Source + 6;
unsigned int tmp;
```

```
tmp = (c[1] << 8) | c[0];
```

```
width = tmp & 0x3FFF;
horizontal_scale = tmp >> 14;
```

```
tmp = (c[3] << 8) | c[2];
```

```
height = tmp & 0x3FFF;
vertical_scale = tmp >> 14;
```

---- End code block -----

[19.2.](#) Frame Header

-----+-----+
Frame Header Type

+-----+-----+		
if (key_frame) {		
color_space	L(1)	
clamping_type	L(1)	
}		
segmentation_enabled	L(1)	
if (segmentation_enabled) {		
update_segmentation()		
}		
filter_type	L(1)	
loop_filter_level	L(6)	
sharpness_level	L(3)	
mb_lf_adjustments()		
log2_nbr_of_dct_partitions	L(2)	
quant_indices()		
if (key_frame) {		
refresh_entropy_probs	L(1)	
} else {		
refresh_golden_frame	L(1)	
refresh_alternate_frame	L(1)	
if (!refresh_golden_frame)		
copy_buffer_to_golden	L(2)	
if (!refresh_alternate_frame)		

copy_buffer_to_alternate	L(2)	
sign_bias_golden	L(1)	
sign_bias_alternate	L(1)	
refresh_entropy_probs	L(1)	
refresh_last	L(1)	
}		
token_prob_update()		
mb_no_coeff_skip	L(1)	
+-----+	+-----+	+

Frame Header	Type
prob_skip_false	L(8)
if (!key_frame) {	
prob_intra	L(8)
prob_last	L(8)
prob_golden	L(8)
intra_16x16_prob_update_flag	L(1)
if (intra_16x16_prob_update_flag) {	
for (i = 0; i < 4; i++)	
intra_16x16_prob	L(8)
}	
intra_chroma_prob_update_flag	L(1)
if (intra_chroma_prob_update_flag) {	
for (i = 0; i < 3; i++)	
intra_chroma_prob	L(8)
}	
mv_prob_update()	
}	

- o color_space defines the YUV color space of the sequence ([Section 9.2](#))

- o `clamping_type` specifies if the decoder is required to clamp the reconstructed pixel values ([Section 9.2](#))
- o `segmentation_enabled` enables the segmentation feature for the current frame ([Section 9.3](#))

- o `filter_type` determines whether the normal or the simple loop filter is used ([Section 9.4](#), [Section 15](#))
- o `loop_filter_level` controls the deblocking filter ([Section 9.4](#), [Section 15](#))
- o `sharpness_level` controls the deblocking filter ([Section 9.4](#), [Section 15](#))
- o `log2_nbr_of_dct_partitions` determines the number of separate partitions containing the DCT coefficients of the macroblocks ([Section 9.5](#))
- o `refresh_entropy_probs` determines whether updated token probabilities are used only for this frame or until further update
- o `refresh_golden_frame` determines if the current decoded frame refreshes the golden frame ([Section 9.7](#))
- o `refresh_alternate_frame` determines if the current decoded frame refreshes the alternate reference frame ([Section 9.7](#))
- o `copy_buffer_to_golden` determines if the golden reference is replaced by another reference ([Section 9.7](#))
- o `copy_buffer_to_alternate` determines if the alternate reference is replaced by another reference ([Section 9.7](#))
- o `sign_bias_golden` controls the sign of motion vectors when the golden frame is referenced ([Section 9.7](#))
- o `sign_bias_alternate` controls the sign of motion vectors when the alternate frame is referenced ([Section 9.7](#))

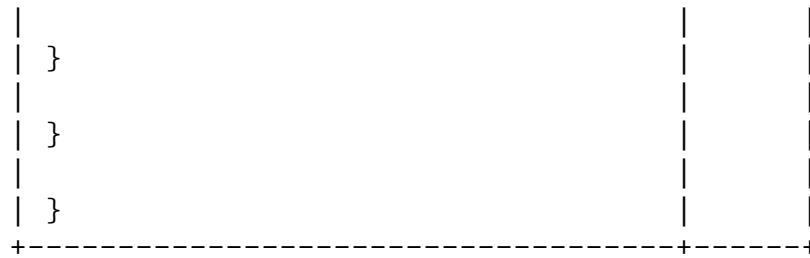
- o refresh_last determines if the current decoded frame refreshes the last frame reference buffer ([Section 9.8](#))
- o mb_no_coeff_skip enables or disables the skipping of macroblocks containing no non-zero coefficients ([Section 9.10](#))
- o prob_skip_false the probability that the macroblock is not skipped (flag indicating skipped macroblock is false) ([Section 9.10](#))
- o prob_intra the probability of an intra macroblock ([Section 9.10](#))
- o prob_last the probability that the last reference frame is used for inter prediction ([Section 9.10](#))

- o prob_golden the probability that the golden reference frame is used for inter prediction ([Section 9.10](#))
- o intra_16x16_prob_update_flag indicates if the branch probabilities used in the decoding of luma intra prediction mode are updated ([Section 9.10](#))
- o intra_16x16_prob the branch probabilities of the luma intra prediction mode decoding tree
- o intra_chroma_prob_update_flag indicates if the branch probabilities used in the decoding of chroma intra prediction mode are updated ([Section 9.10](#))
- o intra_chroma_prob the branch probabilities of the chroma intra prediction mode decoding tree

+-----+-----+	
update_segmentation()	Type
+-----+-----+	
update_mb_segmentation_map	L(1)
update_segment_feature_data	L(1)
if (update_segment_feature_data) {	

segment_feature_mode	L(1)	
for (i = 0; i < 4; i++) {		
quantizer_update	L(1)	
if (quantizer_update) {		
quantizer_update_value	L(7)	
quantizer_update_sign	L(1)	
}		
}		
for (i = 0; i < 4; i++) {		
loop_filter_update	L(1)	
if (loop_filter_update) {		

lf_update_value	L(6)	
lf_update_sign	L(1)	
}		
}		
}		
if (update_mb_segmentation_map) {		
for (i = 0; i < 3; i++) {		
segment_prob_update	L(1)	
if (segment_prob_update) {		
segment_prob	L(8)	



- o `update_mb_segmentation_map` determines if the MB segmentation map is updated in the current frame ([Section 9.3](#))
- o `update_segment_feature_data` indicates if the segment feature data is updated in the current frame ([Section 9.3](#))
- o `segment_feature_mode` indicates the feature data update mode, 0 for delta and 1 for the absolute value ([Section 9.3](#))
- o `quantizer_update` indicates if the quantizer value is updated for the i^{th} segment ([Section 9.3](#))
- o `quantizer_update_value` indicates the update value for the segment quantizer ([Section 9.3](#))
- o `quantizer_update_sign` indicates the update sign for the segment quantizer ([Section 9.3](#))
- o `loop_filter_update` indicates if the loop filter level value is updated for the i^{th} segment ([Section 9.3](#))

- o `lf_update_value` indicates the update value for the loop filter level ([Section 9.3](#))
- o `lf_update_sign` indicates the update sign for the loop filter level ([Section 9.3](#))
- o `segment_prob_update` indicates if the branch probabilities used to decode the `segment_id` in the MB header are decoded from the stream or use the default value of 255 ([Section 9.3](#))
- o `segment_prob` the branch probabilities of the `segment_id` decoding tree ([Section 9.3](#))

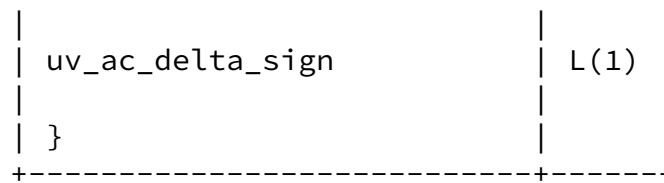
+-----+-----+	
mb_lf_adjustments()	Type
+-----+-----+	
loop_filter_adj_enable	L(1)

if (loop_filter_adj_enable) {		
mode_ref_lf_delta_update	L(1)	
if (mode_ref_lf_delta_update) {		
for (i = 0; i < 4; i++) {		
ref_frame_delta_update_flag	L(1)	
if (ref_frame_delta_update_flag) {		
delta_magnitude	L(6)	
delta_sign	L(1)	
}		
}		
for (i = 0; i < 4; i++) {		
mb_mode_delta_update_flag	L(1)	
if (mb_mode_delta_update_flag) {		
delta_magnitude	L(6)	
delta_sign	L(1)	
}		
}		
}		
}		
+-----+-----+		

- o loop_filter_adj_enable indicates if the MB-level loop filter adjustment (based on the used reference frame and coding mode) is on for the current frame ([Section 9.4](#))

- o mode_ref_lf_delta_update indicates if the delta values used in adjustment are updated in the current frame ([Section 9.4](#))
- o ref_frame_delta_update_flag indicates if the adjustment delta value corresponding to a certain used reference frame is updated ([Section 9.4](#))
- o delta_magnitude is the absolute value of the delta value
- o delta_sign is the sign of the delta value
- o mb_mode_delta_update_flag indicates if the adjustment delta value corresponding to certain MB prediction mode is updated ([Section 9.4](#))

quant_indices()	Type
y_ac_qi	L(7)
y_dc_delta_present	L(1)
if (y_dc_delta_present) {	
y_dc_delta_magnitude	L(4)
y_dc_delta_sign	L(1)
}	
if (y2_dc_delta_present) {	
y2_dc_delta_magnitude	L(4)
y2_dc_delta_sign	L(1)
}	
if (y2_ac_delta_present) {	
y2_ac_delta_magnitude	L(4)
y2_ac_delta_sign	L(1)
}	
if (uv_dc_delta_present) {	
uv_dc_delta_magnitude	L(4)
uv_dc_delta_sign	L(1)
}	
if (uv_ac_delta_present) {	
uv_ac_delta_magnitude	L(4)



- o y_ac_qi is the dequantization table index used for the luma AC coefficients (and other coefficient groups if no delta value is present) ([Section 9.6](#))
- o y_dc_delta_present indicates if the stream contains a delta value that is added to the baseline index to obtain the luma DC coefficient dequantization index ([Section 9.6](#))
- o y_dc_delta_magnitude the magnitude of the delta value ([Section 9.6](#))
- o y_dc_delta_sign the sign of the delta value ([Section 9.6](#))
- o y2_dc_delta_present indicates if the stream contains a delta value that is added to the baseline index to obtain the Y2 block DC coefficient dequantization index ([Section 9.6](#))
- o y2_ac_delta_present indicates if the stream contains a delta value that is added to the baseline index to obtain the Y2 block AC coefficient dequantization index ([Section 9.6](#))
- o uv_dc_delta_present indicates if the stream contains a delta value that is added to the baseline index to obtain the chroma DC coefficient dequantization index ([Section 9.6](#))
- o uv_ac_delta_present indicates if the stream contains a delta value that is added to the baseline index to obtain the chroma AC coefficient dequantization index ([Section 9.6](#))

token_prob_update()	Type
for (i = 0; i < 4; i++) {	
for (j = 0; j < 8; j++) {	
for (k = 0; k < 3; k++) {	
for (l = 0; l < 11; l++) {	
coeff_prob_update_flag	L(1)
if (coeff_prob_update_flag) {	
coeff_prob	L(8)
}	
}	
}	
}	
}	

- o `coeff_prob_update_flag` indicates if the corresponding branch probability is updated in the current frame ([Section 13.4](#))
- o `coeff_prob` is the new branch probability ([Section 13.4](#))

mv_prob_update()	Type
for (i = 0; i < 2; i++) {	
for (j = 0; j < 19; j++) {	
mv_prob_update_flag	L(1)
if (mv_prob_update_flag) {	
prob	L(7)
}	
}	
}	

- o `mv_prob_update_flag` indicates if the corresponding MV decoding

probability is updated in the current frame ([Section 17.2](#))

- o prob is the updated probability ([Section 17.2](#))

[19.3](#). Macroblock Data

Macroblock Data	Type
macroblock_header()	
residual_data()	

macroblock_header()	Type
if (segmentation_map_update) {	
segment_id	T
if (mb_no_coeff_skip) {	
mb_coeff_skip	B(p)
}	

if (!key_frame) {	
is_inter_mb	B(p)
if (is_inter_mb) {	
mb_ref_frame_sel1	B(p)
if (mb_ref_frame_sel1)	
mb_ref_frame_sel2	B(p)
mv_mode	T

if (mv_mode == SPLITMV) {		
mv_split_mode	T	
for (i = 0; i < numMvs; i++) {		
sub_mv_mode	T	
if (sub_mv_mode == NEWMV4x4) {		
read_mvcomponent()		
read_mvcomponent()		
}		
}		
} else if (mv_mode == NEWMV) {		
read_mvcomponent()		
read_mvcomponent()		
}		
} else { /* intra mb */		
intra_y_mode	T	
+-----+	+-----+	

+-----+	+-----+	
macroblock_header()	Type	
+-----+	+-----+	
if (intra_y_mode == B_PRED) {		
for (i = 0; i < 16; i++)		

intra_b_mode	T	
}		
intra_uv_mode	T	
}		
+-----+	+-----+	+-----+

- o segment_id indicates to which segment the macroblock belongs ([Section 10](#))
- o mb_coeff_skip indicates if the macroblock contains any coded coefficients or not ([Section 11.1](#))
- o is_inter_mb indicates if the macroblock is intra or inter coded ([Section 16](#))
- o mb_ref_frame_sel1 selects the reference frame to be used; last frame (0), golden/alternate (1) ([Section 16.2](#))
- o mb_ref_frame_sel2 selects whether the golden (0) or alternate reference frame (1) is used ([Section 16.2](#))
- o mv_mode determines the macroblock motion vector mode ([Section 16.2](#))
- o mv_split_mode gives macroblock partitioning specification and determines number of motion vectors used (numMvs) ([Section 16.2](#))
- o sub_mv_mode determines the sub-macroblock motion vector mode for macroblocks coded using SPLITMV motion vector mode ([Section 16.2](#))
- o intra_y_mode selects the luminance intra prediction mode ([Section 16.1](#))
- o intra_b_mode selects the sub-macroblock luminance prediction mode for macroblocks coded using B_PRED mode ([Section 16.1](#))
- o intra_uv_mode selects the chrominance intra prediction mode ([Section 16.1](#))

residual_data()	Type
<pre> if (!mb_coeff_skip) { if ((is_inter_mb && mv_mode != SPLITMV) (!is_inter_mb && intra_y_mode != B_PRED)) { residual_block() /* Y2 */ } for (i = 0; i < 24; i++) residual_block() /* 16 Y, 4 U, 4 V */ } </pre>	

residual_block()	Type
<pre> for (i = firstCoeff; i < 16; i++) { token if (token == EOB) break; if (token_has_extra_bits) { extra_bits sign } } </pre>	<pre> T L(n) L(1) </pre>

- o firstCoeff is 1 for luma blocks of macroblocks containing Y2 subblock, otherwise 0
- o token defines the value of the coefficient, the value range of the coefficient or the end of block ([Section 13.2](#))
- o extra_bits determine the value of the coefficient within the value range defined by token ([Section 13.2](#))
- o sign indicates the sign of the coefficient ([Section 13.2](#))

[20.](#) Attachment One: Reference Decoder Source Code

Note that the code in this attachment may exhibit bugs, and should be considered a draft until this document reaches RFC status.

[20.1.](#) bit_ops.h

```
---- Begin code block -----

/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */
#ifndef BIT_OPS_H
#define BIT_OPS_H

/* Evaluates to a mask with n bits set */
#define BITS_MASK(n) ((1<<(n))-1)

/* Returns len bits, with the LSB at position bit */
#define BITS_GET(val, bit, len) (((val)>>(bit))&BITS_MASK(len))

#endif

---- End code block -----
```

[20.2.](#) bool_decoder.h

```
---- Begin code block -----

/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license and
 * patent grant that can be found in the LICENSE file in the root of
 * the source tree. All contributing project authors may be found in
 * the AUTHORS file in the root of the source tree.

```

```
*/
```

```
#ifndef BOOL_DECODER_H  
#define BOOL_DECODER_H
```

```
#include <stddef.h>
```

```
struct bool_decoder
```

```
{  
    const unsigned char *input;    /* next compressed data byte */  
    size_t               input_len; /* length of the input buffer */  
    unsigned int         range;     /* identical to encoder's  
                                   * range */  
    unsigned int         value;     /* contains at least 8  
                                   * significant bits */  
    int                  bit_count; /* # of bits shifted out of  
                                   * value, max 7 */  
};
```

```
static void
```

```
init_bool_decoder(struct bool_decoder *d,  
                  const unsigned char *start_partition,  
                  size_t               sz)
```

```
{  
    if (sz >= 2)  
    {  
        d->value = (start_partition[0] << 8) /* first 2 input  
                                             * bytes */  
                | start_partition[1];  
        d->input = start_partition + 2;      /* ptr to next byte */  
        d->input_len = sz - 2;  
    }  
    else  
    {  
        d->value = 0;  
        d->input = NULL;  
        d->input_len = 0;  
    }  
  
    d->range = 255;    /* initial range is full */
```



```

    d->bit_count = 0; /* have not yet shifted out any bits */
}

static int bool_get(struct bool_decoder *d, int probability)
{
    /* range and split are identical to the corresponding values
       used by the encoder when this bool was written */

    unsigned int split = 1 + (((d->range - 1) * probability) >> 8);
    unsigned int 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;

            if (d->input_len)
            {
                d->value |= *d->input++;
                d->input_len--;
            }
        }
    }

```

```

    }

    return retval;
}

static int bool_get_bit(struct bool_decoder *br)
{
    return bool_get(br, 128);
}

static int bool_get_uint(struct bool_decoder *br, int bits)
{
    int z = 0;
    int bit;

    for (bit = bits - 1; bit >= 0; bit--)
    {

```

```

        z |= (bool_get_bit(br) << bit);
    }

    return z;
}

static int bool_get_int(struct bool_decoder *br, int bits)
{
    int z = 0;
    int bit;

    for (bit = bits - 1; bit >= 0; bit--)
    {
        z |= (bool_get_bit(br) << bit);
    }

    return bool_get_bit(br) ? -z : z;
}

static int bool_maybe_get_int(struct bool_decoder *br, int bits)

```

```
{
    return bool_get_bit(br) ? bool_get_int(br, bits) : 0;
}
```

```
static int
bool_read_tree(struct bool_decoder *bool,
               const int          *t,
               const unsigned char *p)
{
    int i = 0;

    while ((i = t[ i + bool_get(bool, p[i>>1])]) > 0) ;

    return -i;
}
#endif
```

---- End code block -----

[20.3.](#) dequant_data.h

---- Begin code block -----

```
static const int dc_q_lookup[128] =
{
    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_q_lookup[128] =
    {
        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
    };

```

---- End code block -----

[20.4.](#) dixie.c

---- Begin code block -----

```

/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be

```

```

* found in the file PATENTS. All contributing project authors may
* be found in the AUTHORS file in the root of the source tree.
*/
#include "vpx_codec_internal.h"
#include "bit_ops.h"
#include "dixie.h"
#include "vp8_prob_data.h"
#include "dequant_data.h"
#include "modemv.h"
#include "tokens.h"
#include "predict.h"
#include "dixie_loopfilter.h"
#include <string.h>
#include <assert.h>

enum
{
    FRAME_HEADER_SZ = 3,
    KEYFRAME_HEADER_SZ = 7
};

#define ARRAY_COPY(a,b) {\
    assert(sizeof(a)==sizeof(b));memcpy(a,b,sizeof(a));}
static void
decode_entropy_header(struct vp8_decoder_ctx      *ctx,
                     struct bool_decoder         *bool,
                     struct vp8_entropy_hdr      *hdr)
{
    int i, j, k, l;

    /* Read coefficient probability updates */
    for (i = 0; i < BLOCK_TYPES; i++)
        for (j = 0; j < COEF_BANDS; j++)
            for (k = 0; k < PREV_COEF_CONTEXTS; k++)
                for (l = 0; l < ENTROPY_NODES; l++)
                    if (bool_get(bool,
                                k_coeff_entropy_update_probs
                                [i][j][k][l]))
                        hdr->coeff_probs[i][j][k][l] =

```

```
bool_get_uint(bool, 8);
```

```

/* Read coefficient skip mode probability */
hdr->coeff_skip_enabled = bool_get_bit(bool);

if (hdr->coeff_skip_enabled)
    hdr->coeff_skip_prob = bool_get_uint(bool, 8);

/* Parse interframe probability updates */
if (!ctx->frame_hdr.is_keyframe)
{
    hdr->prob_inter = bool_get_uint(bool, 8);
    hdr->prob_last  = bool_get_uint(bool, 8);
    hdr->prob_gf    = bool_get_uint(bool, 8);

    if (bool_get_bit(bool))
        for (i = 0; i < 4; i++)
            hdr->y_mode_probs[i] = bool_get_uint(bool, 8);

    if (bool_get_bit(bool))
        for (i = 0; i < 3; i++)
            hdr->uv_mode_probs[i] = bool_get_uint(bool, 8);

    for (i = 0; i < 2; i++)
        for (j = 0; j < MV_PROB_CNT; j++)
            if (bool_get(bool, k_mv_entropy_update_probs[i][j]))
            {
                int x = bool_get_uint(bool, 7);
                hdr->mv_probs[i][j] = x ? x << 1 : 1;
            }
}
}

static void
decode_reference_header(struct vp8_decoder_ctx *ctx,
                      struct bool_decoder *bool,
                      struct vp8_reference_hdr *hdr)
{
    unsigned int key = ctx->frame_hdr.is_keyframe;

    hdr->refresh_gf    = key ? 1 : bool_get_bit(bool);
    hdr->refresh_arf    = key ? 1 : bool_get_bit(bool);
    hdr->copy_gf        = key ? 0 : !hdr->refresh_gf
                          ? bool_get_uint(bool, 2) : 0;
    hdr->copy_arf        = key ? 0 : !hdr->refresh_arf
                          ? bool_get_uint(bool, 2) : 0;
    hdr->sign_bias[GOLDEN_FRAME] = key ? 0 : bool_get_bit(bool);
}

```

```
hdr->sign_bias[ALTREF_FRAME] = key ? 0 : bool_get_bit(bool);
hdr->refresh_entropy = bool_get_bit(bool);
hdr->refresh_last = key ? 1 : bool_get_bit(bool);
}
```

```
static void
decode_quantizer_header(struct vp8_decoder_ctx    *ctx,
                        struct bool_decoder        *bool,
                        struct vp8_quant_hdr       *hdr)
{
    int update;
    int last_q = hdr->q_index;

    hdr->q_index = bool_get_uint(bool, 7);
    update = last_q != hdr->q_index;
    update |= (hdr->y1_dc_delta_q = bool_maybe_get_int(bool, 4));
    update |= (hdr->y2_dc_delta_q = bool_maybe_get_int(bool, 4));
    update |= (hdr->y2_ac_delta_q = bool_maybe_get_int(bool, 4));
    update |= (hdr->uv_dc_delta_q = bool_maybe_get_int(bool, 4));
    update |= (hdr->uv_ac_delta_q = bool_maybe_get_int(bool, 4));
    hdr->delta_update = update;
}
```

```
static void
decode_and_init_token_partitions(struct vp8_decoder_ctx    *ctx,
                                struct bool_decoder        *bool,
                                const unsigned char        *data,
                                unsigned int                sz,
                                struct vp8_token_hdr       *hdr)
{
    int i;

    hdr->partitions = 1 << bool_get_uint(bool, 2);

    if (sz < 3 * (hdr->partitions - 1))
        vpx_internal_error(&ctx->error, VPX_CODEC_CORRUPT_FRAME,
                           "Truncated packet found parsing partition"
                           " lengths.");

    sz -= 3 * (hdr->partitions - 1);

    for (i = 0; i < hdr->partitions; i++)
    {
        if (i < hdr->partitions - 1)
```

```
{
    hdr->partition_sz[i] = (data[2] << 16)
```

```
                                | (data[1] << 8) | data[0];
    data += 3;
}
else
    hdr->partition_sz[i] = sz;

    if (sz < hdr->partition_sz[i])
        vpx_internal_error(&ctx->error, VPX_CODEC_CORRUPT_FRAME,
                           "Truncated partition %d", i);

    sz -= hdr->partition_sz[i];
}

for (i = 0; i < ctx->token_hdr.partitions; i++)
{
    init_bool_decoder(&ctx->tokens[i].bool, data,
                     ctx->token_hdr.partition_sz[i]);
    data += ctx->token_hdr.partition_sz[i];
}
}

static void
decode_loopfilter_header(struct vp8_decoder_ctx    *ctx,
                        struct bool_decoder        *bool,
                        struct vp8_loopfilter_hdr *hdr)
{
    if (ctx->frame_hdr.is_keyframe)
        memset(hdr, 0, sizeof(*hdr));

    hdr->use_simple    = bool_get_bit(bool);
    hdr->level         = bool_get_uint(bool, 6);
    hdr->sharpness     = bool_get_uint(bool, 3);
    hdr->delta_enabled = bool_get_bit(bool);

    if (hdr->delta_enabled && bool_get_bit(bool))
    {
        int i;
```



```

        for (i = 0; i < BLOCK_CONTEXTS; i++)
            hdr->ref_delta[i] = bool_maybe_get_int(bool, 6);

        for (i = 0; i < BLOCK_CONTEXTS; i++)
            hdr->mode_delta[i] = bool_maybe_get_int(bool, 6);
    }
}

```

```

static void
decode_segmentation_header(struct vp8_decoder_ctx *ctx,
                           struct bool_decoder *bool,
                           struct vp8_segment_hdr *hdr)
{
    if (ctx->frame_hdr.is_keyframe)
        memset(hdr, 0, sizeof(*hdr));

    hdr->enabled = bool_get_bit(bool);

    if (hdr->enabled)
    {
        int i;

        hdr->update_map = bool_get_bit(bool);
        hdr->update_data = bool_get_bit(bool);

        if (hdr->update_data)
        {
            hdr->abs = bool_get_bit(bool);

            for (i = 0; i < MAX_MB_SEGMENTS; i++)
                hdr->quant_idx[i] = bool_maybe_get_int(bool, 7);

            for (i = 0; i < MAX_MB_SEGMENTS; i++)
                hdr->lf_level[i] = bool_maybe_get_int(bool, 6);
        }

        if (hdr->update_map)
        {
            for (i = 0; i < MB_FEATURE_TREE_PROBS; i++)
                hdr->tree_probs[i] = bool_get_bit(bool);
        }
    }
}

```

```

        ? bool_get_uint(bool, 8)
        : 255;
    }
}
else
{
    hdr->update_map = 0;
    hdr->update_data = 0;
}
}

static void
dequant_global_init(struct dequant_factors dqf[MAX_MB_SEGMENTS])
{
    int i;

```

```

    for (i = 0; i < MAX_MB_SEGMENTS; i++)
        dqf[i].quant_idx = -1;
}

static int
clamp_q(int q)
{
    if (q < 0) return 0;
    else if (q > 127) return 127;

    return q;
}

static int
dc_q(int q)
{
    return dc_q_lookup[clamp_q(q)];
}

static int
ac_q(int q)
{

```

```

    return ac_q_lookup[clamp_q(q)];
}

static void
dequant_init(struct dequant_factors      factors[MAX_MB_SEGMENTS],
             const struct vp8_segment_hdr *seg,
             const struct vp8_quant_hdr  *quant_hdr)
{
    int i, q;
    struct dequant_factors *dqf = factors;

    for (i = 0; i < (seg->enabled ? MAX_MB_SEGMENTS : 1); i++)
    {
        q = quant_hdr->q_index;

        if (seg->enabled)
            q = (!seg->abs) ? q + seg->quant_idx[i]
                          : seg->quant_idx[i];

        if (dqf->quant_idx != q || quant_hdr->delta_update)
        {
            dqf->factor[TOKEN_BLOCK_Y1][0] =

```

```

        dc_q(q + quant_hdr->y1_dc_delta_q);
dqf->factor[TOKEN_BLOCK_Y1][1] =
    ac_q(q);
dqf->factor[TOKEN_BLOCK_UV][0] =
    dc_q(q + quant_hdr->uv_dc_delta_q);
dqf->factor[TOKEN_BLOCK_UV][1] =
    ac_q(q + quant_hdr->uv_ac_delta_q);
dqf->factor[TOKEN_BLOCK_Y2][0] =
    dc_q(q + quant_hdr->y2_dc_delta_q) * 2;
dqf->factor[TOKEN_BLOCK_Y2][1] =
    ac_q(q + quant_hdr->y2_ac_delta_q) * 155 / 100;

    if (dqf->factor[TOKEN_BLOCK_Y2][1] < 8)
        dqf->factor[TOKEN_BLOCK_Y2][1] = 8;

    if (dqf->factor[TOKEN_BLOCK_UV][0] > 132)
        dqf->factor[TOKEN_BLOCK_UV][0] = 132;

```

```

        dqf->quant_idx = q;
    }

    dqf++;
}

}

static void
decode_frame(struct vp8_decoder_ctx *ctx,
             const unsigned char *data,
             unsigned int sz)
{
    vpx_codec_err_t res;
    struct bool_decoder bool;
    int i, row, partition;

    ctx->saved_entropy_valid = 0;

    if ((res = vp8_parse_frame_header(data, sz, &ctx->frame_hdr)))
        vpx_internal_error(&ctx->error, res,
                          "Failed to parse frame header");

    if (ctx->frame_hdr.is_experimental)
        vpx_internal_error(&ctx->error, VPX_CODEC_UNSUP_BITSTREAM,
                          "Experimental bitstreams not supported.");

    data += FRAME_HEADER_SZ;
    sz -= FRAME_HEADER_SZ;

```

```

if (ctx->frame_hdr.is_keyframe)
{
    data += KEYFRAME_HEADER_SZ;
    sz -= KEYFRAME_HEADER_SZ;
    ctx->mb_cols = (ctx->frame_hdr.kf.w + 15) / 16;
    ctx->mb_rows = (ctx->frame_hdr.kf.h + 15) / 16;
}

/* Start the bitreader for the header/entropy partition */
init_bool_decoder(&bool, data, ctx->frame_hdr.part0_sz);

```

```

/* Skip the colorspace and clamping bits */
if (ctx->frame_hdr.is_keyframe)
    if (bool_get_uint(&bool, 2))
        vpx_internal_error(
            &ctx->error, VPX_CODEC_UNSUP_BITSTREAM,
            "Reserved bits not supported.");

decode_segmentation_header(ctx, &bool, &ctx->segment_hdr);
decode_loopfilter_header(ctx, &bool, &ctx->loopfilter_hdr);
decode_and_init_token_partitions(ctx,
                                &bool,
                                data + ctx->frame_hdr.part0_sz,
                                sz - ctx->frame_hdr.part0_sz,
                                &ctx->token_hdr);
decode_quantizer_header(ctx, &bool, &ctx->quant_hdr);
decode_reference_header(ctx, &bool, &ctx->reference_hdr);

/* Set keyframe entropy defaults. These get updated on keyframes
 * regardless of the refresh_entropy setting.
 */
if (ctx->frame_hdr.is_keyframe)
{
    ARRAY_COPY(ctx->entropy_hdr.coeff_probs,
                k_default_coeff_probs);
    ARRAY_COPY(ctx->entropy_hdr.mv_probs,
                k_default_mv_probs);
    ARRAY_COPY(ctx->entropy_hdr.y_mode_probs,
                k_default_y_mode_probs);
    ARRAY_COPY(ctx->entropy_hdr.uv_mode_probs,
                k_default_uv_mode_probs);
}

if (!ctx->reference_hdr.refresh_entropy)
{
    ctx->saved_entropy = ctx->entropy_hdr;
    ctx->saved_entropy_valid = 1;
}

```

```

decode_entropy_header(ctx, &bool, &ctx->entropy_hdr);

vp8_dixie_modemv_init(ctx);
vp8_dixie_tokens_init(ctx);

```

```

vp8_dixie_predict_init(ctx);
dequant_init(ctx->dequant_factors, &ctx->segment_hdr,
             &ctx->quant_hdr);

for (row = 0, partition = 0; row < ctx->mb_rows; row++)
{
    vp8_dixie_modemv_process_row(
        ctx, &bool, row, 0, ctx->mb_cols);
    vp8_dixie_tokens_process_row(ctx, partition, row, 0,
                                ctx->mb_cols);
    vp8_dixie_predict_process_row(ctx, row, 0, ctx->mb_cols);

    if (ctx->loopfilter_hdr.level && row)
        vp8_dixie_loopfilter_process_row(ctx, row - 1, 0,
                                           ctx->mb_cols);

    if (++partition == ctx->token_hdr.partitions)
        partition = 0;
}

if (ctx->loopfilter_hdr.level)
    vp8_dixie_loopfilter_process_row(
        ctx, row - 1, 0, ctx->mb_cols);

ctx->frame_cnt++;

if (!ctx->reference_hdr.refresh_entropy)
{
    ctx->entropy_hdr = ctx->saved_entropy;
    ctx->saved_entropy_valid = 0;
}

/* Handle reference frame updates */
if (ctx->reference_hdr.copy_arf == 1)
{
    vp8_dixie_release_ref_frame(ctx->ref_frames[ALTREF_FRAME]);
    ctx->ref_frames[ALTREF_FRAME] =
        vp8_dixie_ref_frame(ctx->ref_frames[LAST_FRAME]);
}
else if (ctx->reference_hdr.copy_arf == 2)
{
    vp8_dixie_release_ref_frame(ctx->ref_frames[ALTREF_FRAME]);
    ctx->ref_frames[ALTREF_FRAME] =
        vp8_dixie_ref_frame(ctx->ref_frames[GOLDEN_FRAME]);
}

```

```
    }

    if (ctx->reference_hdr.copy_gf == 1)
    {
        vp8_dixie_release_ref_frame(ctx->ref_frames[GOLDEN_FRAME]);
        ctx->ref_frames[GOLDEN_FRAME] =
            vp8_dixie_ref_frame(ctx->ref_frames[LAST_FRAME]);
    }
    else if (ctx->reference_hdr.copy_gf == 2)
    {
        vp8_dixie_release_ref_frame(ctx->ref_frames[GOLDEN_FRAME]);
        ctx->ref_frames[GOLDEN_FRAME] =
            vp8_dixie_ref_frame(ctx->ref_frames[ALTREF_FRAME]);
    }

    if (ctx->reference_hdr.refresh_gf)
    {
        vp8_dixie_release_ref_frame(ctx->ref_frames[GOLDEN_FRAME]);
        ctx->ref_frames[GOLDEN_FRAME] =
            vp8_dixie_ref_frame(ctx->ref_frames[CURRENT_FRAME]);
    }

    if (ctx->reference_hdr.refresh_arf)
    {
        vp8_dixie_release_ref_frame(ctx->ref_frames[ALTREF_FRAME]);
        ctx->ref_frames[ALTREF_FRAME] =
            vp8_dixie_ref_frame(ctx->ref_frames[CURRENT_FRAME]);
    }

    if (ctx->reference_hdr.refresh_last)
    {
        vp8_dixie_release_ref_frame(ctx->ref_frames[LAST_FRAME]);
        ctx->ref_frames[LAST_FRAME] =
            vp8_dixie_ref_frame(ctx->ref_frames[CURRENT_FRAME]);
    }
}

void
vp8_dixie_decode_init(struct vp8_decoder_ctx *ctx)
{
    dequant_global_init(ctx->dequant_factors);
}

#define CHECK_FOR_UPDATE(lval,rval,update_flag) do {\
```

```
    unsigned int old = lval; \
```

```
        update_flag |= (old != (lval = rval)); \
    } while(0)

vpx_codec_err_t
vp8_parse_frame_header(const unsigned char  *data,
                      unsigned int          sz,
                      struct vp8_frame_hdr  *hdr)
{
    unsigned long raw;

    if (sz < 10)
        return VPX_CODEC_CORRUPT_FRAME;

    /* The frame header is defined as a three byte little endian
     * value
     */
    raw = data[0] | (data[1] << 8) | (data[2] << 16);
    hdr->is_keyframe      = !BITS_GET(raw, 0, 1);
    hdr->version          = BITS_GET(raw, 1, 2);
    hdr->is_experimental  = BITS_GET(raw, 3, 1);
    hdr->is_shown         = BITS_GET(raw, 4, 1);
    hdr->part0_sz         = BITS_GET(raw, 5, 19);

    if (sz <= hdr->part0_sz + (hdr->is_keyframe ? 10 : 3))
        return VPX_CODEC_CORRUPT_FRAME;

    hdr->frame_size_updated = 0;

    if (hdr->is_keyframe)
    {
        unsigned int update = 0;

        /* Keyframe header consists of a three byte sync code
         * followed by the width and height and associated scaling
         * factors.
         */
        if (data[3] != 0x9d || data[4] != 0x01 || data[5] != 0x2a)
            return VPX_CODEC_UNSUP_BITSTREAM;

        raw = data[6] | (data[7] << 8)
```



```

        | (data[8] << 16) | (data[9] << 24);
CHECK_FOR_UPDATE(hdr->kf.w,      BITS_GET(raw,  0, 14),
                  update);
CHECK_FOR_UPDATE(hdr->kf.scale_w, BITS_GET(raw, 14,  2),
                  update);
CHECK_FOR_UPDATE(hdr->kf.h,      BITS_GET(raw, 16, 14),
                  update);
CHECK_FOR_UPDATE(hdr->kf.scale_h, BITS_GET(raw, 30,  2),

```

```

        update);

    hdr->frame_size_updated = update;

    if (!hdr->kf.w || !hdr->kf.h)
        return VPX_CODEC_UNSUP_BITSTREAM;
}

return VPX_CODEC_OK;
}

vpx_codec_err_t
vp8_dixie_decode_frame(struct vp8_decoder_ctx *ctx,
                      const unsigned char *data,
                      unsigned int        sz)
{
    volatile struct vp8_decoder_ctx *ctx_ = ctx;

    ctx->error.error_code = VPX_CODEC_OK;
    ctx->error.has_detail = 0;

    if (!setjmp(ctx->error.jmp))
        decode_frame(ctx, data, sz);

    return ctx_>error.error_code;
}

void
vp8_dixie_decode_destroy(struct vp8_decoder_ctx *ctx)
{
    vp8_dixie_predict_destroy(ctx);
}

```

```

    vp8_dixie_tokens_destroy(ctx);
    vp8_dixie_modemv_destroy(ctx);
}

```

---- End code block -----

[20.5.](#) dixie.h

---- Begin code block -----

```

/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license

```

```

 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */
#ifndef DIXIE_H
#define DIXIE_H
#include "vpx_codec_internal.h"
#include "bool_decoder.h"

struct vp8_frame_hdr
{
    unsigned int is_keyframe;      /* Frame is a keyframe */
    unsigned int is_experimental; /* Frame is a keyframe */
    unsigned int version;          /* Bitstream version */
    unsigned int is_shown;         /* Frame is to be displayed. */
    unsigned int part0_sz;         /* Partition 0 length, in bytes */

    struct vp8_kf_hdr
    {
        unsigned int w;           /* Width */
        unsigned int h;           /* Height */
        unsigned int scale_w;     /* Scaling factor, Width */
        unsigned int scale_h;     /* Scaling factor, Height */
    } kf;

```

```

        unsigned int frame_size_updated; /* Flag to indicate a resolution
                                           * update.
                                           */
    };

enum
{
    MB_FEATURE_TREE_PROBS = 3,
    MAX_MB_SEGMENTS = 4
};

struct vp8_segment_hdr
{
    unsigned int    enabled;
    unsigned int    update_data;
    unsigned int    update_map;
    unsigned int    abs; /* 0=deltas, 1=absolute values */
    unsigned int    tree_probs[MB_FEATURE_TREE_PROBS];
    int             lf_level[MAX_MB_SEGMENTS];
    int             quant_idx[MAX_MB_SEGMENTS];
};

```

```

    };

enum
{
    BLOCK_CONTEXTS = 4
};

struct vp8_loopfilter_hdr
{
    unsigned int    use_simple;
    unsigned int    level;
    unsigned int    sharpness;
    unsigned int    delta_enabled;
    int             ref_delta[BLOCK_CONTEXTS];
    int             mode_delta[BLOCK_CONTEXTS];
};

```

```

enum
{
    MAX_PARTITIONS = 8
};

struct vp8_token_hdr
{
    unsigned int    partitions;
    unsigned int    partition_sz[MAX_PARTITIONS];
};

struct vp8_quant_hdr
{
    unsigned int    q_index;
    int             delta_update;
    int             y1_dc_delta_q;
    int             y2_dc_delta_q;
    int             y2_ac_delta_q;
    int             uv_dc_delta_q;
    int             uv_ac_delta_q;
};

struct vp8_reference_hdr
{
    unsigned int refresh_last;
    unsigned int refresh_gf;

```

```

    unsigned int refresh_arf;
    unsigned int copy_gf;
    unsigned int copy_arf;
    unsigned int sign_bias[4];
    unsigned int refresh_entropy;
};

enum
{
    BLOCK_TYPES          = 4,
    PREV_COEF_CONTEXTS = 3,

```

```

        COEF_BANDS          = 8,
        ENTROPY_NODES       = 11,
    };
    typedef unsigned char coeff_probs_table_t[BLOCK_TYPES][COEF_BANDS]
    [PREV_COEF_CONTEXTS]
    [ENTROPY_NODES];

    enum
    {
        MV_PROB_CNT = 2 + 8 - 1 + 10 /* from entropymv.h */
    };
    typedef unsigned char mv_component_probs_t[MV_PROB_CNT];

    struct vp8_entropy_hdr
    {
        coeff_probs_table_t    coeff_probs;
        mv_component_probs_t    mv_probs[2];
        unsigned int           coeff_skip_enabled;
        unsigned char          coeff_skip_prob;
        unsigned char          y_mode_probs[4];
        unsigned char          uv_mode_probs[3];
        unsigned char          prob_inter;
        unsigned char          prob_last;
        unsigned char          prob_gf;
    };

    enum reference_frame
    {
        CURRENT_FRAME,
        LAST_FRAME,
        GOLDEN_FRAME,
        ALTREF_FRAME,
        NUM_REF_FRAMES
    };

```

```
};
```

```
enum prediction_mode
{
```

```

/* 16x16 intra modes */
DC_PRED, V_PRED, H_PRED, TM_PRED, B_PRED,

/* 16x16 inter modes */
NEARESTMV, NEARMV, ZEROMV, NEWMV, SPLITMV,

MB_MODE_COUNT,

/* 4x4 intra modes */
B_DC_PRED = 0, B_TM_PRED, B_VE_PRED, B_HE_PRED, B_LD_PRED,
B_RD_PRED, B_VR_PRED, B_VL_PRED, B_HD_PRED, B_HU_PRED,

/* 4x4 inter modes */
LEFT4X4, ABOVE4X4, ZERO4X4, NEW4X4,

B_MODE_COUNT
};

enum splitmv_partitioning
{
    SPLITMV_16X8,
    SPLITMV_8X16,
    SPLITMV_8X8,
    SPLITMV_4X4
};

typedef short filter_t[6];

typedef union mv
{
    struct
    {
        int16_t x, y;
    } d;
    uint32_t raw;
} mv_t;

struct mb_base_info
{

```

[illegible]

```
struct ref_cnt_img
{
    vpx_image_t  img;
    unsigned int ref_cnt;
};

struct vp8_decoder_ctx
{
    struct vpx_internal_error_info error;
    unsigned int                    frame_cnt;

    struct vp8_frame_hdr           frame_hdr;
    struct vp8_segment_hdr         segment_hdr;
    struct vp8_loopfilter_hdr      loopfilter_hdr;
    struct vp8_token_hdr           token_hdr;
    struct vp8_quant_hdr           quant_hdr;
    struct vp8_reference_hdr       reference_hdr;
    struct vp8_entropy_hdr         entropy_hdr;

    struct vp8_entropy_hdr         saved_entropy;
    unsigned int                   saved_entropy_valid;

    unsigned int                   mb_rows;
    unsigned int                   mb_cols;
    struct mb_info                 *mb_info_storage;
    struct mb_info                 **mb_info_rows_storage;
    struct mb_info                 **mb_info_rows;

    token_entropy_ctx_t            *above_token_entropy_ctx;
    struct token_decoder            tokens[MAX_PARTITIONS];
    struct dequant_factors          dequant_factors[MAX_MB_SEGMENTS];

    struct ref_cnt_img             frame_strg[NUM_REF_FRAMES];
    struct ref_cnt_img             *ref_frames[NUM_REF_FRAMES];
    ptrdiff_t                      ref_frame_offsets[4];

    const filter_t                 *subpixel_filters;
};

void
```



```
vp8_dixie_decode_init(struct vp8_decoder_ctx *ctx);
```

```
void  
vp8_dixie_decode_destroy(struct vp8_decoder_ctx *ctx);
```

```
vpx_codec_err_t  
vp8_parse_frame_header(const unsigned char *data,  
                        unsigned int      sz,  
                        struct vp8_frame_hdr *hdr);
```

```
vpx_codec_err_t  
vp8_dixie_decode_frame(struct vp8_decoder_ctx *ctx,  
                       const unsigned char *data,  
                       unsigned int      sz);
```

```
#define CLAMP_255(x) ((x)<0?0:((x)>255?255:(x)))
```

```
#endif
```

```
---- End code block -----
```

[20.6.](#) dixie_loopfilter.c

```
---- Begin code block -----
```

```
/*  
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.  
 *  
 * Use of this source code is governed by a BSD-style license  
 * that can be found in the LICENSE file in the root of the source  
 * tree. An additional intellectual property rights grant can be  
 * found in the file PATENTS. All contributing project authors may  
 * be found in the AUTHORS file in the root of the source tree.  
 */  
#include "dixie.h"  
#include "dixie_loopfilter.h"
```

```
#define ABS(x) ((x) >= 0 ? (x) : -(x))
```

```
#define p3 pixels[-4*stride]
#define p2 pixels[-3*stride]
#define p1 pixels[-2*stride]
#define p0 pixels[-1*stride]
#define q0 pixels[ 0*stride]
#define q1 pixels[ 1*stride]
#define q2 pixels[ 2*stride]
#define q3 pixels[ 3*stride]
```

```
#define static
static int
```

```
saturate_int8(int x)
{
    if (x < -128)
        return -128;

    if (x > 127)
        return 127;

    return x;
}
```

```
static int
saturate_uint8(int x)
{
    if (x < 0)
        return 0;

    if (x > 255)
        return 255;

    return x;
}
```

```
static int
high_edge_variance(unsigned char *pixels,
                   int stride,
```

```

        int                hev_threshold)
{
    return ABS(p1 - p0) > hev_threshold ||
           ABS(q1 - q0) > hev_threshold;
}

static int
simple_threshold(unsigned char *pixels,
                int           stride,
                int           filter_limit)
{
    return (ABS(p0 - q0) * 2 + (ABS(p1 - q1) >> 1)) <= filter_limit;
}

static int
normal_threshold(unsigned char *pixels,
                int           stride,
                int           edge_limit,

```

```

        int                interior_limit)
{
    int E = edge_limit;
    int I = interior_limit;

    return simple_threshold(pixels, stride, 2 * E + I)
           && ABS(p3 - p2) <= I && ABS(p2 - p1) <= I
           && ABS(p1 - p0) <= I && ABS(q3 - q2) <= I
           && ABS(q2 - q1) <= I && ABS(q1 - q0) <= I;
}

static void
filter_common(unsigned char *pixels,
                int           stride,
                int           use_outer_taps)
{
    int a, f1, f2;

    a = 3 * (q0 - p0);

```

```

    if (use_outer_taps)
        a += saturate_int8(p1 - q1);

    a = saturate_int8(a);

    f1 = ((a + 4 > 127) ? 127 : a + 4) >> 3;
    f2 = ((a + 3 > 127) ? 127 : a + 3) >> 3;

    p0 = saturate_uint8(p0 + f2);
    q0 = saturate_uint8(q0 - f1);

    if (!use_outer_taps)
    {
        /* This handles the case of subblock_filter()
         * (from the bitstream guide.
         */
        a = (f1 + 1) >> 1;
        p1 = saturate_uint8(p1 + a);
        q1 = saturate_uint8(q1 - a);
    }
}

static void
filter_mb_edge(unsigned char *pixels,
               int           stride)
{

```

```

    int w, a;

    w = saturate_int8(saturate_int8(p1 - q1) + 3 * (q0 - p0));

    a = (27 * w + 63) >> 7;
    p0 = saturate_uint8(p0 + a);
    q0 = saturate_uint8(q0 - a);

    a = (18 * w + 63) >> 7;
    p1 = saturate_uint8(p1 + a);
    q1 = saturate_uint8(q1 - a);

    a = (9 * w + 63) >> 7;
    p2 = saturate_uint8(p2 + a);

```

```

        q2 = saturate_uint8(q2 - a);
    }

static void
filter_mb_v_edge(unsigned char *src,
                 int          stride,
                 int          edge_limit,
                 int          interior_limit,
                 int          hev_threshold,
                 int          size)
{
    int i;

    for (i = 0; i < 8 * size; i++)
    {
        if (normal_threshold(src, 1, edge_limit, interior_limit))
        {
            if (high_edge_variance(src, 1, hev_threshold))
                filter_common(src, 1, 1);
            else
                filter_mb_edge(src, 1);
        }

        src += stride;
    }
}

static void
filter_subblock_v_edge(unsigned char *src,
                       int          stride,
                       int          edge_limit,

```

```

                                int          interior_limit,
                                int          hev_threshold,
                                int          size)
{
    int i;

    for (i = 0; i < 8 * size; i++)

```

```

    {
        if (normal_threshold(src, 1, edge_limit, interior_limit))
            filter_common(src, 1,
                          high_edge_variance(src, 1, hev_threshold));

        src += stride;
    }
}

static void
filter_mb_h_edge(unsigned char *src,
                 int          stride,
                 int          edge_limit,
                 int          interior_limit,
                 int          hev_threshold,
                 int          size)
{
    int i;

    for (i = 0; i < 8 * size; i++)
    {
        if (normal_threshold(src, stride, edge_limit, interior_limit))
        {
            if (high_edge_variance(src, stride, hev_threshold))
                filter_common(src, stride, 1);
            else
                filter_mb_edge(src, stride);
        }

        src += 1;
    }
}

static void
filter_subblock_h_edge(unsigned char *src,
                      int          stride,
                      int          edge_limit,
                      int          interior_limit,
                      int          hev_threshold,

```

```

                                int          size)
{
    int i;

    for (i = 0; i < 8 * size; i++)
    {
        if (normal_threshold(src, stride, edge_limit, interior_limit))
            filter_common(src, stride,
                          high_edge_variance(src, stride,
                                              hev_threshold));

        src += 1;
    }
}

```

```

static void
filter_v_edge_simple(unsigned char *src,
                     int          stride,
                     int          filter_limit)
{
    int i;

    for (i = 0; i < 16; i++)
    {
        if (simple_threshold(src, 1, filter_limit))
            filter_common(src, 1, 1);

        src += stride;
    }
}

```

```

static void
filter_h_edge_simple(unsigned char *src,
                     int          stride,
                     int          filter_limit)
{
    int i;

    for (i = 0; i < 16; i++)
    {
        if (simple_threshold(src, stride, filter_limit))
            filter_common(src, stride, 1);

        src += 1;
    }
}

```

```
static void
calculate_filter_parameters(struct vp8_decoder_ctx *ctx,
                           struct mb_info         *mbi,
                           int                     *edge_limit_,
                           int                     *interior_limit_,
                           int                     *hev_threshold_)
{
    int filter_level, interior_limit, hev_threshold;

    /* Reference code/spec seems to conflate filter_level and
     * edge_limit
     */

    filter_level = ctx->loopfilter_hdr.level;

    if (ctx->segment_hdr.enabled)
    {
        if (!ctx->segment_hdr.abs)
            filter_level +=
                ctx->segment_hdr.lf_level[mbi->base.segment_id];
        else
            filter_level =
                ctx->segment_hdr.lf_level[mbi->base.segment_id];
    }

    if (ctx->loopfilter_hdr.delta_enabled)
    {
        filter_level +=
            ctx->loopfilter_hdr.ref_delta[mbi->base.ref_frame];

        if (mbi->base.ref_frame == CURRENT_FRAME)
        {
            if (mbi->base.y_mode == B_PRED)
                filter_level += ctx->loopfilter_hdr.mode_delta[0];
        }
        else if (mbi->base.y_mode == ZEROMV)
            filter_level += ctx->loopfilter_hdr.mode_delta[1];
        else if (mbi->base.y_mode == SPLITMV)
            filter_level += ctx->loopfilter_hdr.mode_delta[3];
        else
            filter_level += ctx->loopfilter_hdr.mode_delta[2];
    }
}
```



```

if (filter_level > 63)
    filter_level = 63;
else if (filter_level < 0)
    filter_level = 0;

```

```

interior_limit = filter_level;

if (ctx->loopfilter_hdr.sharpness)
{
    interior_limit >>= ctx->loopfilter_hdr.sharpness > 4 ? 2 : 1;

    if (interior_limit > 9 - ctx->loopfilter_hdr.sharpness)
        interior_limit = 9 - ctx->loopfilter_hdr.sharpness;
}

if (interior_limit < 1)
    interior_limit = 1;

hev_threshold = (filter_level >= 15);

if (filter_level >= 40)
    hev_threshold++;

if (filter_level >= 20 && !ctx->frame_hdr.is_keyframe)
    hev_threshold++;

*edge_limit_ = filter_level;
*interior_limit_ = interior_limit;
*hev_threshold_ = hev_threshold;
}

static void
filter_row_normal(struct vp8_decoder_ctx *ctx,
                 unsigned int      row,
                 unsigned int      start_col,
                 unsigned int      num_cols)
{
    unsigned char *y, *u, *v;
    int          stride, uv_stride;
    struct mb_info *mbi;

```

```

unsigned int    col;

/* Adjust pointers based on row, start_col */
stride    = ctx->ref_frames[CURRENT_FRAME]->img.stride[PLANE_Y];
uv_stride = ctx->ref_frames[CURRENT_FRAME]->img.stride[PLANE_U];
y = ctx->ref_frames[CURRENT_FRAME]->img.planes[PLANE_Y];
u = ctx->ref_frames[CURRENT_FRAME]->img.planes[PLANE_U];
v = ctx->ref_frames[CURRENT_FRAME]->img.planes[PLANE_V];
y += (stride * row + start_col) * 16;
u += (uv_stride * row + start_col) * 8;
v += (uv_stride * row + start_col) * 8;
mbi = ctx->mb_info_rows[row] + start_col;

```

```

for (col = start_col; col < start_col + num_cols; col++)
{
    int edge_limit, interior_limit, hev_threshold;

    /* TODO: only need to recalculate every MB if segmentation is
     * enabled.
     */
    calculate_filter_parameters(ctx, mbi, &edge_limit,
                               &interior_limit, &hev_threshold);

    if (edge_limit)
    {
        if (col)
        {
            filter_mb_v_edge(y, stride, edge_limit + 2,
                             interior_limit, hev_threshold, 2);
            filter_mb_v_edge(u, uv_stride, edge_limit + 2,
                             interior_limit, hev_threshold, 1);
            filter_mb_v_edge(v, uv_stride, edge_limit + 2,
                             interior_limit, hev_threshold, 1);
        }

        /* NOTE: This conditional is actually dependent on the
         * number of coefficients decoded, not the skip flag as
         * coded in the bitstream. The tokens task is expected to
         * set 31 if there is *any* non-zero data.
         */
        if (mbi->base.eob_mask
            || mbi->base.y_mode == SPLITMV

```



```

        hev_threshold, 2);
    filter_subblock_h_edge(u + 4 * uv_stride, uv_stride,
        edge_limit, interior_limit,
        hev_threshold, 1);
    filter_subblock_h_edge(v + 4 * uv_stride, uv_stride,
        edge_limit, interior_limit,
        hev_threshold, 1);
    }
}

y += 16;
u += 8;
v += 8;
mbi++;
}
}

```

```

static void
filter_row_simple(struct vp8_decoder_ctx *ctx,
    unsigned int      row,
    unsigned int      start_col,
    unsigned int      num_cols)
{
    unsigned char *y;
    int          stride;

```

```

struct mb_info *mbi;
unsigned int    col;

/* Adjust pointers based on row, start_col */
stride = ctx->ref_frames[CURRENT_FRAME]->img.stride[PLANE_Y];
y = ctx->ref_frames[CURRENT_FRAME]->img.planes[PLANE_Y];
y += (stride * row + start_col) * 16;
mbi = ctx->mb_info_rows[row] + start_col;

for (col = start_col; col < start_col + num_cols; col++)
{
    int edge_limit, interior_limit, hev_threshold;

    /* TODO: only need to recalculate every MB if segmentation is
     * enabled.

```

```

    */
    calculate_filter_parameters(ctx, mbi, &edge_limit,
                              &interior_limit, &hev_threshold);

    if (edge_limit)
    {
        /* NOTE: This conditional is actually dependent on the
         * number of coefficients decoded, not the skip flag as
         * coded in the bitstream. The tokens task is expected to
         * set 31 if there is *any* non-zero data.
         */
        int filter_subblocks = (mbi->base.eob_mask
                                || mbi->base.y_mode == SPLITMV
                                || mbi->base.y_mode == B_PRED);
        int mb_limit = (edge_limit + 2) * 2 + interior_limit;
        int b_limit = edge_limit * 2 + interior_limit;

        if (col)
            filter_v_edge_simple(y, stride, mb_limit);

        if (filter_subblocks)
        {
            filter_v_edge_simple(y + 4, stride, b_limit);
            filter_v_edge_simple(y + 8, stride, b_limit);
            filter_v_edge_simple(y + 12, stride, b_limit);
        }

        if (row)
            filter_h_edge_simple(y, stride, mb_limit);

        if (filter_subblocks)
        {

```

```

            filter_h_edge_simple(y + 4 * stride, stride, b_limit);
            filter_h_edge_simple(y + 8 * stride, stride, b_limit);
            filter_h_edge_simple(y + 12 * stride, stride, b_limit);
        }
    }

    y += 16;
    mbi++;

```

```

    }
}

void
vp8_dixie_loopfilter_process_row(struct vp8_decoder_ctx *ctx,
                                unsigned int          row,
                                unsigned int          start_col,
                                unsigned int          num_cols)
{
    if (ctx->loopfilter_hdr.use_simple)
        filter_row_simple(ctx, row, start_col, num_cols);
    else
        filter_row_normal(ctx, row, start_col, num_cols);
}

```

---- End code block -----

---- Begin code block -----

```
/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */
#ifndef DIXIE_LOOPFILTER_H
#define DIXIE_LOOPFILTER_H

void
vp8_dixie_loopfilter_process_row(struct vp8_decoder_ctx *ctx,
                                unsigned int          row,
                                unsigned int          start_col,
                                unsigned int          num_cols);

#endif
```

---- End code block -----

[20.8.](#) idct_add.c

---- Begin code block -----

```
/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */
#include "dixie.h"
#include "idct_add.h"
#include <assert.h>

void
vp8_dixie_walsh(const short *input, short *output)
{
    int i;
```

```
int a1, b1, c1, d1;
int a2, b2, c2, d2;
const short *ip = input;
short *op = output;

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;
}
}
```



```
#define cospi8sqrt2minus1 20091
#define sinpi8sqrt2      35468
```

```
#define rounding          0
static void
idct_columns(const short *input, short *output)
{
    int i;
    int a1, b1, c1, d1;

    const short *ip = input;
    short *op = output;
    int temp1, temp2;
    int shortpitch = 4;

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

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

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

```

void
vp8_dixie_idct_add(unsigned char      *recon,
                  const unsigned char *predict,
                  int                  stride,
                  const short          *coeffs)
{
    int i;
    int a1, b1, c1, d1, temp1, temp2;
    short tmp[16];

```

```

    idct_columns(coeffs, tmp);
    coeffs = tmp;

    for (i = 0; i < 4; i++)
    {
        a1 = coeffs[0] + coeffs[2];
        b1 = coeffs[0] - coeffs[2];

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

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

        recon[0] = CLAMP_255(predict[0] + ((a1 + d1 + 4) >> 3));
        recon[3] = CLAMP_255(predict[3] + ((a1 - d1 + 4) >> 3));
        recon[1] = CLAMP_255(predict[1] + ((b1 + c1 + 4) >> 3));
        recon[2] = CLAMP_255(predict[2] + ((b1 - c1 + 4) >> 3));

        coeffs += 4;
        recon += stride;
        predict += stride;
    }
}

```

----- End code block -----

[20.9.](#) idct_add.h

----- Begin code block -----

```
/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */
#ifndef IDCT_ADD_H
#define IDCT_ADD_H

void
vp8_dixie_idct_add_init(struct vp8_decoder_ctx *ctx);

void
vp8_dixie_idct_add(unsigned char      *recon,
                  const unsigned char *predict,
                  int                  stride,
                  const short          *coeffs);
```

```

void
vp8_dixie_walsh(const short *in, short *out);

void
vp8_dixie_idct_add_process_row(struct vp8_decoder_ctx *ctx,
                               short                  *coeffs,
                               unsigned int            row,
                               unsigned int            start_col,
                               unsigned int            num_cols);

#endif

```

---- End code block -----

[20.10.](#) mem.h

---- Begin code block -----

```
/*
```

Bankoski, et al. Expires November 19, 2011 [Page 186]

Internet-Draft VP8 Data Format and Decoding Guide May 2011

```

* Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
*
* Use of this source code is governed by a BSD-style license
* that can be found in the LICENSE file in the root of the source
* tree. An additional intellectual property rights grant can be
* found in the file PATENTS. All contributing project authors may
* be found in the AUTHORS file in the root of the source tree.
*/

```

```

#ifndef VPX_PORTS_MEM_H
#define VPX_PORTS_MEM_H
#include "vpx_config.h"
#include "vpx_integer.h"

#if defined(__GNUC__) && __GNUC__
#define DECLARE_ALIGNED(n,typ,val)  typ val __attribute__ \
    ((aligned (n)))

```

```

#elif defined(_MSC_VER)
#define DECLARE_ALIGNED(n,typ,val)  __declspec(align(n)) typ val
#else
#warning No alignment directives known for this compiler.
#define DECLARE_ALIGNED(n,typ,val)  typ val
#endif
#endif

/* Declare an aligned array on the stack, for situations where the
 * stack pointer may not have the alignment we expect. Creates an
 * array with a modified name, then defines val to be a pointer, and
 * aligns that pointer within the array.
 */
#define DECLARE_ALIGNED_ARRAY(a,typ,val,n)\
typ val##_[(n)+(a)/sizeof(typ)+1];\
typ *val = (typ*)((((intptr_t)val##_)+(a)-1)&((intptr_t)-(a)))

/* Indicates that the usage of the specified variable has been
 * audited to assure that it's safe to use uninitialized. Silences
 * 'may be used uninitialized' warnings on gcc.
 */
#if defined(__GNUC__) && __GNUC__
#define UNINITIALIZED_IS_SAFE(x) x=x
#else
#define UNINITIALIZED_IS_SAFE(x) x
#endif

```

---- End code block -----

[20.11.](#) modemv.c

---- Begin code block -----

```

/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source

```

```

* tree. An additional intellectual property rights grant can be
* found in the file PATENTS. All contributing project authors may
* be found in the AUTHORS file in the root of the source tree.
*/

```

```

#include "dixie.h"
#include "modemv_data.h"
#include <stdlib.h>
#include <assert.h>

```

```

struct mv_clamp_rect
{
    int to_left, to_right, to_top, to_bottom;
};

```

```

static union mv
    clamp_mv(union mv raw, const struct mv_clamp_rect *bounds)
{
    union mv newmv;

    newmv.d.x = (raw.d.x < bounds->to_left)
        ? bounds->to_left : raw.d.x;
    newmv.d.x = (raw.d.x > bounds->to_right)
        ? bounds->to_right : newmv.d.x;
    newmv.d.y = (raw.d.y < bounds->to_top)
        ? bounds->to_top : raw.d.y;
    newmv.d.y = (raw.d.y > bounds->to_bottom)
        ? bounds->to_bottom : newmv.d.y;
    return newmv;
}

```

```

static int
read_segment_id(struct bool_decoder *bool,
                struct vp8_segment_hdr *seg)
{

```

```

    return bool_get(bool, seg->tree_probs[0])
        ? 2 + bool_get(bool, seg->tree_probs[2])
        : bool_get(bool, seg->tree_probs[1]);
}

```

```

static enum prediction_mode
above_block_mode(const struct mb_info *this,
                  const struct mb_info *above,
                  unsigned int b)
{
    if (b < 4)
    {
        switch (above->base.y_mode)
        {
            case DC_PRED:
                return B_DC_PRED;
            case V_PRED:
                return B_VE_PRED;
            case H_PRED:
                return B_HE_PRED;
            case TM_PRED:
                return B_TM_PRED;
            case B_PRED:
                return above->split.modes[b+12];
            default:
                assert(0);
        }
    }

    return this->split.modes[b-4];
}

```

```

static enum prediction_mode
left_block_mode(const struct mb_info *this,
                 const struct mb_info *left,
                 unsigned int b)
{
    if (!(b & 3))
    {
        switch (left->base.y_mode)
        {
            case DC_PRED:
                return B_DC_PRED;
            case V_PRED:
                return B_VE_PRED;
            case H_PRED:

```

```
        return B_HE_PRED;
    case TM_PRED:
        return B_TM_PRED;
    case B_PRED:
        return left->split.modes[b+3];
    default:
        assert(0);
    }
}

return this->split.modes[b-1];
}

static void
decode_kf_mb_mode(struct mb_info      *this,
                  struct mb_info      *left,
                  struct mb_info      *above,
                  struct bool_decoder *bool)
{
    int y_mode, uv_mode;

    y_mode = bool_read_tree(bool, kf_y_mode_tree, kf_y_mode_probs);

    if (y_mode == B_PRED)
    {
        unsigned int i;

        for (i = 0; i < 16; i++)
        {
            enum prediction_mode a = above_block_mode(this, above, i);
            enum prediction_mode l = left_block_mode(this, left, i);
            enum prediction_mode b;

            b = bool_read_tree(bool, b_mode_tree,
                               kf_b_mode_probs[a][l]);
            this->split.modes[i] = b;
        }
    }

    uv_mode = bool_read_tree(bool, uv_mode_tree, kf_uv_mode_probs);

    this->base.y_mode = y_mode;
    this->base.uv_mode = uv_mode;
    this->base.mv.raw = 0;
    this->base.ref_frame = 0;
}
```

```
static void
decode_intra_mb_mode(struct mb_info      *this,
                    struct vp8_entropy_hdr *hdr,
                    struct bool_decoder  *bool)
{
    /* Like decode_kf_mb_mode, but with probabilities transmitted in the
     * bitstream and no context on the above/left block mode.
     */
    int y_mode, uv_mode;

    y_mode = bool_read_tree(bool, y_mode_tree, hdr->y_mode_probs);

    if (y_mode == B_PRED)
    {
        unsigned int i;

        for (i = 0; i < 16; i++)
        {
            enum prediction_mode b;

            b = bool_read_tree(bool, b_mode_tree, default_b_mode_probs);
            this->split.modes[i] = b;
        }
    }

    uv_mode = bool_read_tree(bool, uv_mode_tree, hdr->uv_mode_probs);

    this->base.y_mode = y_mode;
    this->base.uv_mode = uv_mode;
    this->base.mv.raw = 0;
    this->base.ref_frame = CURRENT_FRAME;
}

static int
read_mv_component(struct bool_decoder *bool,
                  const unsigned char mvc[MV_PROB_CNT])
{
    enum {IS_SHORT, SIGN, SHORT, BITS = SHORT + 8 - 1, LONG_WIDTH = 10};
    int x = 0;
```

```

if (bool_get(bool, mvc[IS_SHORT])) /* Large */
{
    int i = 0;

    for (i = 0; i < 3; i++)
        x += bool_get(bool, mvc[BITS + i]) << i;

```

```

/* Skip bit 3, which is sometimes implicit */
for (i = LONG_WIDTH - 1; i > 3; i--)
    x += bool_get(bool, mvc[BITS + i]) << i;

if (!(x & 0xFFF0) || bool_get(bool, mvc[BITS + 3]))
    x += 8;
}
else /* small */
    x = bool_read_tree(bool, small_mv_tree, mvc + SHORT);

if (x && bool_get(bool, mvc[SIGN]))
    x = -x;

return x << 1;
}

```

```

static mv_t
above_block_mv(const struct mb_info *this,
               const struct mb_info *above,
               unsigned int          b)
{
    if (b < 4)
    {
        if (above->base.y_mode == SPLITMV)
            return above->split.mvs[b+12];

        return above->base.mv;
    }

    return this->split.mvs[b-4];
}

```

```

static mv_t
left_block_mv(const struct mb_info *this,
               const struct mb_info *left,
               unsigned int          b)
{
    if (!(b & 3))
    {
        if (left->base.y_mode == SPLITMV)
            return left->split.mvs[b+3];

        return left->base.mv;
    }

    return this->split.mvs[b-1];
}

```

Bankoski, et al.

Expires November 19, 2011

[Page 192]

Internet-Draft

VP8 Data Format and Decoding Guide

May 2011

```

}

```

```

static enum prediction_mode
submv_ref(struct bool_decoder *bool, union mv l, union mv a)
{
    enum subblock_mv_ref
    {
        SUBMVREF_NORMAL,
        SUBMVREF_LEFT_ZED,
        SUBMVREF_ABOVE_ZED,
        SUBMVREF_LEFT_ABOVE_SAME,
        SUBMVREF_LEFT_ABOVE_ZED
    };

    int lez = !(l.raw);
    int aez = !(a.raw);
    int lea = l.raw == a.raw;
    enum subblock_mv_ref ctx = SUBMVREF_NORMAL;

    if (lea && lez)
        ctx = SUBMVREF_LEFT_ABOVE_ZED;
    else if (lea)
        ctx = SUBMVREF_LEFT_ABOVE_SAME;
    else if (aez)
        ctx = SUBMVREF_ABOVE_ZED;
}

```

```

        else if (lez)
            ctx = SUBMVREF_LEFT_ZED;

        return bool_read_tree(bool, submv_ref_tree, submv_ref_probs2[ctx]);
    }

```

```

static void
read_mv(struct bool_decoder *bool,
        union mv            *mv,
        mv_component_probs_t mvc[2])
{
    mv->d.y = read_mv_component(bool, mvc[0]);
    mv->d.x = read_mv_component(bool, mvc[1]);
}

```

```

static void
mv_bias(const struct mb_info *mb,
        const unsigned int  sign_bias[3],
        enum reference_frame ref_frame,
        union mv            *mv)

```

```

{
    if (sign_bias[mb->base.ref_frame] ^ sign_bias[ref_frame])
    {
        mv->d.x *= -1;
        mv->d.y *= -1;
    }
}

```

```

enum near_mv_v
{
    CNT_BEST = 0,
    CNT_ZEROZERO = 0,
    CNT_NEAREST,
    CNT_NEAR,
    CNT_SPLITMV
};

```

```

static void
find_near_mvs(const struct mb_info  *this,
               const struct mb_info  *left,
               const struct mb_info  *above,
               const unsigned int     sign_bias[3],
               union mv               near_mvs[4],
               int                    cnt[4])
{
    const struct mb_info *aboveleft = above - 1;
    union mv             *mv = near_mvs;
    int                  *cntx = cnt;

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

    /* Process above */
    if (above->base.ref_frame != CURRENT_FRAME)
    {
        if (above->base.mv.raw)
        {
            (++mv)->raw = above->base.mv.raw;
            mv_bias(above, sign_bias, this->base.ref_frame, mv);
            ++cntx;
        }

        *cntx += 2;
    }
}

```

```

/* Process left */
if (left->base.ref_frame != CURRENT_FRAME)
{
    if (left->base.mv.raw)
    {
        union mv this_mv;

        this_mv.raw = left->base.mv.raw;
        mv_bias(left, sign_bias, this->base.ref_frame, &this_mv);

        if (this_mv.raw != mv->raw)
        {

```

```

        (++mv)->raw = this_mv.raw;
        ++cntx;
    }

    *cntx += 2;
}
else
    cnt[CNT_ZEROZERO] += 2;
}

/* Process above left */
if (aboveleft->base.ref_frame != CURRENT_FRAME)
{
    if (aboveleft->base.mv.raw)
    {
        union mv this_mv;

        this_mv.raw = aboveleft->base.mv.raw;
        mv_bias(aboveleft, sign_bias, this->base.ref_frame,
                &this_mv);

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

        *cntx += 1;
    }
    else
        cnt[CNT_ZEROZERO] += 1;
}

/* If we have three distinct MV's ... */
if (cnt[CNT_SPLITMV])
{

```

```

    /* See if above-left MV can be merged with NEAREST */
    if (mv->raw == near_mvs[CNT_NEAREST].raw)
        cnt[CNT_NEAREST] += 1;
}

```

```

cnt[CNT_SPLITMV] = ((above->base.y_mode == SPLITMV)
                    + (left->base.y_mode == SPLITMV)) * 2
                    + (aboveleft->base.y_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].raw;
    near_mvs[CNT_NEAREST].raw = near_mvs[CNT_NEAR].raw;
    near_mvs[CNT_NEAR].raw = tmp;
}

/* Use near_mvs[CNT_BEST] to store the "best" MV. Note that this
 * storage shares the same address as near_mvs[CNT_ZEROZERO].
 */
if (cnt[CNT_NEAREST] >= cnt[CNT_BEST])
    near_mvs[CNT_BEST] = near_mvs[CNT_NEAREST];
}

static void
decode_split_mv(struct mb_info      *this,
                const struct mb_info *left,
                const struct mb_info *above,
                struct vp8_entropy_hdr *hdr,
                union mv            *best_mv,
                struct bool_decoder  *bool)
{
    const int *partition;
    int j, k, mask, partition_id;

    partition_id = bool_read_tree(bool, split_mv_tree, split_mv_probs);
    partition = mv_partitions[partition_id];
    this->base.partitioning = partition_id;

    for (j = 0, mask = 0; mask < 65535; j++)
    {
        union mv mv, left_mv, above_mv;
        enum prediction_mode subblock_mode;

```

```

/* Find the first subblock in this partition. */
for (k = 0; j != partition[k]; k++);

/* Decode the next MV */
left_mv = left_block_mv(this, left, k);
above_mv = above_block_mv(this, above, k);
subblock_mode = submv_ref(bool, left_mv, above_mv);

switch (subblock_mode)
{
case LEFT4X4:
    mv = left_mv;
    break;
case ABOVE4X4:
    mv = above_mv;
    break;
case ZERO4X4:
    mv.raw = 0;
    break;
case NEW4X4:
    read_mv(bool, &mv, hdr->mv_probs);
    mv.d.x += best_mv->d.x;
    mv.d.y += best_mv->d.y;
    break;
default:
    assert(0);
}

/* Fill the MV's for this partition */
for (; k < 16; k++)
    if (j == partition[k])
    {
        this->split.mvs[k] = mv;
        mask |= 1 << k;
    }
}

static int
need_mc_border(union mv mv, int l, int t, int b_w, int w, int h)
{
    int b, r;

    /* Get distance to edge for top-left pixel */
    l += (mv.d.x >> 3);
    t += (mv.d.y >> 3);

```



```
/* Get distance to edge for bottom-right pixel */
r = w - (l + b_w);
b = h - (t + b_w);

return (l >> 1 < 2 || r >> 1 < 3 || t >> 1 < 2 || b >> 1 < 3);
}

static void
decode_mvs(struct vp8_decoder_ctx      *ctx,
           struct mb_info              *this,
           const struct mb_info        *left,
           const struct mb_info        *above,
           const struct mv_clamp_rect   *bounds,
           struct bool_decoder         *bool)
{
    struct vp8_entropy_hdr *hdr = &ctx->entropy_hdr;
    union mv                near_mvs[4];
    union mv                clamped_best_mv;
    int                     mv_cnts[4];
    unsigned char           probs[4];
    enum {BEST, NEAREST, NEAR};
    int x, y, w, h, b;

    this->base.ref_frame = bool_get(bool, hdr->prob_last)
        ? 2 + bool_get(bool, hdr->prob_gf)
        : 1;

    find_near_mvs(this, this - 1, above, ctx->reference_hdr.sign_bias,
                  near_mvs, mv_cnts);
    probs[0] = mv_counts_to_probs[mv_cnts[0]][0];
    probs[1] = mv_counts_to_probs[mv_cnts[1]][1];
    probs[2] = mv_counts_to_probs[mv_cnts[2]][2];
    probs[3] = mv_counts_to_probs[mv_cnts[3]][3];

    this->base.y_mode = bool_read_tree(bool, mv_ref_tree, probs);
    this->base.uv_mode = this->base.y_mode;

    this->base.need_mc_border = 0;
    x = (-bounds->to_left - 128) >> 3;
    y = (-bounds->to_top - 128) >> 3;
    w = ctx->mb_cols * 16;
    h = ctx->mb_rows * 16;
```

```

switch (this->base.y_mode)
{
case NEARESTMV:
    this->base.mv = clamp_mv(near_mvs[NEAREST], bounds);
    break;

```

```

case NEARMV:
    this->base.mv = clamp_mv(near_mvs[NEAR], bounds);
    break;
case ZEROMV:
    this->base.mv.raw = 0;
    return; //skip need_mc_border check
case NEWMV:
    clamped_best_mv = clamp_mv(near_mvs[BEST], bounds);
    read_mv(bool, &this->base.mv, hdr->mv_probs);
    this->base.mv.d.x += clamped_best_mv.d.x;
    this->base.mv.d.y += clamped_best_mv.d.y;
    break;
case SPLITMV:
{
    union mv          chroma_mv[4] = {{{0}}};

    clamped_best_mv = clamp_mv(near_mvs[BEST], bounds);
    decode_split_mv(this, left, above, hdr, &clamped_best_mv, bool);
    this->base.mv = this->split.mvs[15];

    for (b = 0; b < 16; b++)
    {
        chroma_mv[(b>>1&1) + (b>>2&2)].d.x +=
            this->split.mvs[b].d.x;
        chroma_mv[(b>>1&1) + (b>>2&2)].d.y +=
            this->split.mvs[b].d.y;

        if (need_mc_border(this->split.mvs[b],
            x + (b & 3) * 4, y + (b & ~3), 4, w, h))
        {
            this->base.need_mc_border = 1;
            break;
        }
    }
}

```

```

for (b = 0; b < 4; b++)
{
    chroma_mv[b].d.x += 4 + 8 * (chroma_mv[b].d.x >> 31);
    chroma_mv[b].d.y += 4 + 8 * (chroma_mv[b].d.y >> 31);
    chroma_mv[b].d.x /= 4;
    chroma_mv[b].d.y /= 4;

    //note we're passing in non-subsampled coordinates
    if (need_mc_border(chroma_mv[b],
        x + (b & 1) * 8, y + (b >> 1) * 8, 16, w, h))
    {
        this->base.need_mc_border = 1;
        break;
    }
}

```

```

    }
}

return; //skip need_mc_border check
}
default:
    assert(0);
}

if (need_mc_border(this->base.mv, x, y, 16, w, h))
    this->base.need_mc_border = 1;
}

void
vp8_dixie_modemv_process_row(struct vp8_decoder_ctx *ctx,
struct bool_decoder *bool,
int row,
int start_col,
int num_cols)
{
    struct mb_info *above, *this;
    unsigned int col;
    struct mv_clamp_rect bounds;

    this = ctx->mb_info_rows[row] + start_col;
    above = ctx->mb_info_rows[row - 1] + start_col;
}

```

```

/* Calculate the eighth-pel MV bounds using a 1 MB border. */
bounds.to_left   = -((start_col + 1) << 7);
bounds.to_right  = (ctx->mb_cols - start_col) << 7;
bounds.to_top    = -((row + 1) << 7);
bounds.to_bottom = (ctx->mb_rows - row) << 7;

for (col = start_col; col < start_col + num_cols; col++)
{
    if (ctx->segment_hdr.update_map)
        this->base.segment_id = read_segment_id(bool,
            &ctx->segment_hdr);

    if (ctx->entropy_hdr.coeff_skip_enabled)
        this->base.skip_coeff = bool_get(bool,
            ctx->entropy_hdr.coeff_skip_prob);

    if (ctx->frame_hdr.is_keyframe)
    {
        if (!ctx->segment_hdr.update_map)
            this->base.segment_id = 0;
    }
}

```

```

        decode_kf_mb_mode(this, this - 1, above, bool);
    }
    else
    {
        if (bool_get(bool, ctx->entropy_hdr.prob_inter))
            decode_mvs(ctx, this, this - 1, above, &bounds, bool);
        else
            decode_intra_mb_mode(this, &ctx->entropy_hdr, bool);

        bounds.to_left -= 16 << 3;
        bounds.to_right -= 16 << 3;
    }

    /* Advance to next mb */
    this++;
    above++;
}
}

```

void

```

vp8_dixie_modemv_init(struct vp8_decoder_ctx *ctx)
{
    unsigned int    mbi_w, mbi_h, i;
    struct mb_info *mbi;

    mbi_w = ctx->mb_cols + 1; /* For left border col */
    mbi_h = ctx->mb_rows + 1; /* For above border row */

    if (ctx->frame_hdr.frame_size_updated)
    {
        free(ctx->mb_info_storage);
        ctx->mb_info_storage = NULL;
        free(ctx->mb_info_rows_storage);
        ctx->mb_info_rows_storage = NULL;
    }

    if (!ctx->mb_info_storage)
        ctx->mb_info_storage = calloc(mbi_w * mbi_h,
                                       sizeof(*ctx->mb_info_storage));

    if (!ctx->mb_info_rows_storage)
        ctx->mb_info_rows_storage = calloc(mbi_h,
                                           sizeof(*ctx->mb_info_rows_storage));

    /* Set up row pointers */
    mbi = ctx->mb_info_storage + 1;

```

```

    for (i = 0; i < mbi_h; i++)
    {
        ctx->mb_info_rows_storage[i] = mbi;
        mbi += mbi_w;
    }

    ctx->mb_info_rows = ctx->mb_info_rows_storage + 1;
}

void
vp8_dixie_modemv_destroy(struct vp8_decoder_ctx *ctx)
{
    free(ctx->mb_info_storage);

```

```
ctx->mb_info_storage = NULL;
free(ctx->mb_info_rows_storage);
ctx->mb_info_rows_storage = NULL;
}
```

---- End code block -----

[20.12.](#) modenv.h

---- Begin code block -----

```
/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
```

```

* Use of this source code is governed by a BSD-style license
* that can be found in the LICENSE file in the root of the source
* tree. An additional intellectual property rights grant can be
* found in the file PATENTS. All contributing project authors may
* be found in the AUTHORS file in the root of the source tree.
*/
#ifndef MODEMV_H
#define MODEMV_H

void
vp8_dixie_modemv_init(struct vp8_decoder_ctx *ctx);

void
vp8_dixie_modemv_destroy(struct vp8_decoder_ctx *ctx);

void
vp8_dixie_modemv_process_row(struct vp8_decoder_ctx *ctx,
                             struct bool_decoder *bool,
                             int row,
                             int start_col,
                             int num_cols);

#endif

---- End code block -----

```

[20.13.](#) modemv_data.h

```

---- Begin code block -----

static const unsigned char kf_y_mode_probs[] = { 145, 156, 163, 128};
static const unsigned char kf_uv_mode_probs[] = { 142, 114, 183};
static const unsigned char kf_b_mode_probs[10][10][9] =
{
    { /* above mode 0 */
        { /* left mode 0 */ 231, 120, 48, 89, 115, 113, 120, 152, 112},
        { /* left mode 1 */ 152, 179, 64, 126, 170, 118, 46, 70, 95},
        { /* left mode 2 */ 175, 69, 143, 80, 85, 82, 72, 155, 103},
    }
}

```

```

{ /* left mode 3 */ 56, 58, 10, 171, 218, 189, 17, 13, 152},
{ /* left mode 4 */ 144, 71, 10, 38, 171, 213, 144, 34, 26},
{ /* left mode 5 */ 114, 26, 17, 163, 44, 195, 21, 10, 173},
{ /* left mode 6 */ 121, 24, 80, 195, 26, 62, 44, 64, 85},
{ /* left mode 7 */ 170, 46, 55, 19, 136, 160, 33, 206, 71},
{ /* left mode 8 */ 63, 20, 8, 114, 114, 208, 12, 9, 226},
{ /* left mode 9 */ 81, 40, 11, 96, 182, 84, 29, 16, 36}
},
{ /* above mode 1 */
{ /* left mode 0 */ 134, 183, 89, 137, 98, 101, 106, 165, 148},
{ /* left mode 1 */ 72, 187, 100, 130, 157, 111, 32, 75, 80},
{ /* left mode 2 */ 66, 102, 167, 99, 74, 62, 40, 234, 128},
{ /* left mode 3 */ 41, 53, 9, 178, 241, 141, 26, 8, 107},
{ /* left mode 4 */ 104, 79, 12, 27, 217, 255, 87, 17, 7},
{ /* left mode 5 */ 74, 43, 26, 146, 73, 166, 49, 23, 157},
{ /* left mode 6 */ 65, 38, 105, 160, 51, 52, 31, 115, 128},
{ /* left mode 7 */ 87, 68, 71, 44, 114, 51, 15, 186, 23},
{ /* left mode 8 */ 47, 41, 14, 110, 182, 183, 21, 17, 194},
{ /* left mode 9 */ 66, 45, 25, 102, 197, 189, 23, 18, 22}
},
{ /* above mode 2 */
{ /* left mode 0 */ 88, 88, 147, 150, 42, 46, 45, 196, 205},
{ /* left mode 1 */ 43, 97, 183, 117, 85, 38, 35, 179, 61},
{ /* left mode 2 */ 39, 53, 200, 87, 26, 21, 43, 232, 171},
{ /* left mode 3 */ 56, 34, 51, 104, 114, 102, 29, 93, 77},
{ /* left mode 4 */ 107, 54, 32, 26, 51, 1, 81, 43, 31},
{ /* left mode 5 */ 39, 28, 85, 171, 58, 165, 90, 98, 64},
{ /* left mode 6 */ 34, 22, 116, 206, 23, 34, 43, 166, 73},
{ /* left mode 7 */ 68, 25, 106, 22, 64, 171, 36, 225, 114},
{ /* left mode 8 */ 34, 19, 21, 102, 132, 188, 16, 76, 124},
{ /* left mode 9 */ 62, 18, 78, 95, 85, 57, 50, 48, 51}
},
{ /* above mode 3 */
{ /* left mode 0 */ 193, 101, 35, 159, 215, 111, 89, 46, 111},
{ /* left mode 1 */ 60, 148, 31, 172, 219, 228, 21, 18, 111},
{ /* left mode 2 */ 112, 113, 77, 85, 179, 255, 38, 120, 114},
{ /* left mode 3 */ 40, 42, 1, 196, 245, 209, 10, 25, 109},
{ /* left mode 4 */ 100, 80, 8, 43, 154, 1, 51, 26, 71},
{ /* left mode 5 */ 88, 43, 29, 140, 166, 213, 37, 43, 154},
{ /* left mode 6 */ 61, 63, 30, 155, 67, 45, 68, 1, 209},
{ /* left mode 7 */ 142, 78, 78, 16, 255, 128, 34, 197, 171},
{ /* left mode 8 */ 41, 40, 5, 102, 211, 183, 4, 1, 221},
{ /* left mode 9 */ 51, 50, 17, 168, 209, 192, 23, 25, 82}
},
{ /* above mode 4 */
{ /* left mode 0 */ 125, 98, 42, 88, 104, 85, 117, 175, 82},
{ /* left mode 1 */ 95, 84, 53, 89, 128, 100, 113, 101, 45},
{ /* left mode 2 */ 75, 79, 123, 47, 51, 128, 81, 171, 1},

```



```
{ /* left mode 3 */ 57, 17, 5, 71, 102, 57, 53, 41, 49},
{ /* left mode 4 */ 115, 21, 2, 10, 102, 255, 166, 23, 6},
{ /* left mode 5 */ 38, 33, 13, 121, 57, 73, 26, 1, 85},
{ /* left mode 6 */ 41, 10, 67, 138, 77, 110, 90, 47, 114},
{ /* left mode 7 */ 101, 29, 16, 10, 85, 128, 101, 196, 26},
{ /* left mode 8 */ 57, 18, 10, 102, 102, 213, 34, 20, 43},
{ /* left mode 9 */ 117, 20, 15, 36, 163, 128, 68, 1, 26}
},
{ /* above mode 5 */
{ /* left mode 0 */ 138, 31, 36, 171, 27, 166, 38, 44, 229},
{ /* left mode 1 */ 67, 87, 58, 169, 82, 115, 26, 59, 179},
{ /* left mode 2 */ 63, 59, 90, 180, 59, 166, 93, 73, 154},
{ /* left mode 3 */ 40, 40, 21, 116, 143, 209, 34, 39, 175},
{ /* left mode 4 */ 57, 46, 22, 24, 128, 1, 54, 17, 37},
{ /* left mode 5 */ 47, 15, 16, 183, 34, 223, 49, 45, 183},
{ /* left mode 6 */ 46, 17, 33, 183, 6, 98, 15, 32, 183},
{ /* left mode 7 */ 65, 32, 73, 115, 28, 128, 23, 128, 205},
{ /* left mode 8 */ 40, 3, 9, 115, 51, 192, 18, 6, 223},
{ /* left mode 9 */ 87, 37, 9, 115, 59, 77, 64, 21, 47}
},
{ /* above mode 6 */
{ /* left mode 0 */ 104, 55, 44, 218, 9, 54, 53, 130, 226},
{ /* left mode 1 */ 64, 90, 70, 205, 40, 41, 23, 26, 57},
{ /* left mode 2 */ 54, 57, 112, 184, 5, 41, 38, 166, 213},
{ /* left mode 3 */ 30, 34, 26, 133, 152, 116, 10, 32, 134},
{ /* left mode 4 */ 75, 32, 12, 51, 192, 255, 160, 43, 51},
{ /* left mode 5 */ 39, 19, 53, 221, 26, 114, 32, 73, 255},
{ /* left mode 6 */ 31, 9, 65, 234, 2, 15, 1, 118, 73},
{ /* left mode 7 */ 88, 31, 35, 67, 102, 85, 55, 186, 85},
{ /* left mode 8 */ 56, 21, 23, 111, 59, 205, 45, 37, 192},
{ /* left mode 9 */ 55, 38, 70, 124, 73, 102, 1, 34, 98}
},
{ /* above mode 7 */
{ /* left mode 0 */ 102, 61, 71, 37, 34, 53, 31, 243, 192},
{ /* left mode 1 */ 69, 60, 71, 38, 73, 119, 28, 222, 37},
{ /* left mode 2 */ 68, 45, 128, 34, 1, 47, 11, 245, 171},
{ /* left mode 3 */ 62, 17, 19, 70, 146, 85, 55, 62, 70},
{ /* left mode 4 */ 75, 15, 9, 9, 64, 255, 184, 119, 16},
{ /* left mode 5 */ 37, 43, 37, 154, 100, 163, 85, 160, 1},
{ /* left mode 6 */ 63, 9, 92, 136, 28, 64, 32, 201, 85},
{ /* left mode 7 */ 86, 6, 28, 5, 64, 255, 25, 248, 1},
{ /* left mode 8 */ 56, 8, 17, 132, 137, 255, 55, 116, 128},
{ /* left mode 9 */ 58, 15, 20, 82, 135, 57, 26, 121, 40}
```

```

},
{ /* above mode 8 */
{ /* left mode 0 */ 164, 50, 31, 137, 154, 133, 25, 35, 218},
{ /* left mode 1 */ 51, 103, 44, 131, 131, 123, 31, 6, 158},
{ /* left mode 2 */ 86, 40, 64, 135, 148, 224, 45, 183, 128},

```

```

{ /* left mode 3 */ 22, 26, 17, 131, 240, 154, 14, 1, 209},
{ /* left mode 4 */ 83, 12, 13, 54, 192, 255, 68, 47, 28},
{ /* left mode 5 */ 45, 16, 21, 91, 64, 222, 7, 1, 197},
{ /* left mode 6 */ 56, 21, 39, 155, 60, 138, 23, 102, 213},
{ /* left mode 7 */ 85, 26, 85, 85, 128, 128, 32, 146, 171},
{ /* left mode 8 */ 18, 11, 7, 63, 144, 171, 4, 4, 246},
{ /* left mode 9 */ 35, 27, 10, 146, 174, 171, 12, 26, 128}
},
{ /* above mode 9 */
{ /* left mode 0 */ 190, 80, 35, 99, 180, 80, 126, 54, 45},
{ /* left mode 1 */ 85, 126, 47, 87, 176, 51, 41, 20, 32},
{ /* left mode 2 */ 101, 75, 128, 139, 118, 146, 116, 128, 85},
{ /* left mode 3 */ 56, 41, 15, 176, 236, 85, 37, 9, 62},
{ /* left mode 4 */ 146, 36, 19, 30, 171, 255, 97, 27, 20},
{ /* left mode 5 */ 71, 30, 17, 119, 118, 255, 17, 18, 138},
{ /* left mode 6 */ 101, 38, 60, 138, 55, 70, 43, 26, 142},
{ /* left mode 7 */ 138, 45, 61, 62, 219, 1, 81, 188, 64},
{ /* left mode 8 */ 32, 41, 20, 117, 151, 142, 20, 21, 163},
{ /* left mode 9 */ 112, 19, 12, 61, 195, 128, 48, 4, 24}
}
};
static const int kf_y_mode_tree[] =
{
-B_PRED, 2,
4, 6,
-DC_PRED, -V_PRED,
-H_PRED, -TM_PRED
};
static const int y_mode_tree[] =
{
-DC_PRED, 2,
4, 6,
-V_PRED, -H_PRED,
-TM_PRED, -B_PRED
};
static const int uv_mode_tree[6] =

```

```

{
    -DC_PRED, 2,
    -V_PRED, 4,
    -H_PRED, -TM_PRED
};
static const int b_mode_tree[18] =
{
    -B_DC_PRED, 2,                /* 0 = DC_NODE */
    -B_TM_PRED, 4,                /* 1 = TM_NODE */
    -B_VE_PRED, 6,                /* 2 = VE_NODE */
    8, 12,                        /* 3 = COM_NODE */
    -B_HE_PRED, 10,               /* 4 = HE_NODE */

```

```

    -B_RD_PRED, -B_VR_PRED,        /* 5 = RD_NODE */
    -B_LD_PRED, 14,                /* 6 = LD_NODE */
    -B_VL_PRED, 16,                /* 7 = VL_NODE */
    -B_HD_PRED, -B_HU_PRED         /* 8 = HD_NODE */
};
static const int small_mv_tree[14] =
{
    2, 8,
    4, 6,
    -0, -1,
    -2, -3,
    10, 12,
    -4, -5,
    -6, -7
};
static const int mv_ref_tree[8] =
{
    -ZEROMV, 2,
    -NEARESTMV, 4,
    -NEARMV, 6,
    -NEWMV, -SPLITMV
};
static const int submv_ref_tree[6] =
{
    -LEFT4X4, 2,
    -ABOVE4X4, 4,
    -ZERO4X4, -NEW4X4
};
static const int split_mv_tree[6] =

```

```

{
    -3, 2,
    -2, 4,
    -0, -1
};
static const unsigned char default_b_mode_probs[] =
{ 120, 90, 79, 133, 87, 85, 80, 111, 151};
static const unsigned char mv_counts_to_probs[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 }

};
static const unsigned char split_mv_probs[3] =
{ 110, 111, 150};

```

```

static const unsigned char submv_ref_probs2[5][3] =
{
    { 147, 136, 18 },
    { 106, 145, 1 },
    { 179, 121, 1 },
    { 223, 1, 34 },
    { 208, 1, 1 }
};

const static int mv_partitions[4][16] =
{
    {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1 },
    {0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1 },
    {0, 0, 1, 1, 0, 0, 1, 1, 2, 2, 3, 3, 2, 2, 3, 3 },
    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 }
};

```

---- End code block -----

---- Begin code block -----

```
/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */
#include "dixie.h"
#include "predict.h"
#include "idct_add.h"
#include "mem.h"
#include <assert.h>
#include <string.h>

enum
{
    BORDER_PIXELS      = 16,
};

static const filter_t sixtap_filters[8] =
{
```

```
    { 0,  0, 128,    0,  0,  0 },
    { 0, -6, 123,   12, -1,  0 },
    { 2, -11, 108,   36, -8,  1 },
    { 0, -9,  93,   50, -6,  0 },
    { 3, -16,  77,   77, -16,  3 },
    { 0, -6,  50,   93, -9,  0 },
    { 1, -8,  36,  108, -11,  2 },
    { 0, -1,  12,  123, -6,  0 },
};
```

```
static const filter_t bilinear_filters[8] =
{
    { 0,  0, 128,    0,  0,  0 },
```

```

    { 0, 0, 112, 16, 0, 0 },
    { 0, 0, 96, 32, 0, 0 },
    { 0, 0, 80, 48, 0, 0 },
    { 0, 0, 64, 64, 0, 0 },
    { 0, 0, 48, 80, 0, 0 },
    { 0, 0, 32, 96, 0, 0 },
    { 0, 0, 16, 112, 0, 0 }
};

```

```

static void
predict_h_nxn(unsigned char *predict,
              int           stride,
              int           n)
{
    unsigned char *left = predict - 1;
    int           i, j;

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            predict[i * stride + j] = left[i * stride];
}

```

```

static void
predict_v_nxn(unsigned char *predict,
              int           stride,
              int           n)
{
    unsigned char *above = predict - stride;
    int           i, j;

    for (i = 0; i < n; i++)

```

```

        for (j = 0; j < n; j++)
            predict[i * stride + j] = above[j];
}

```

```

static void
predict_tm_nxn(unsigned char *predict,
               int           stride,

```

```

        int            n)
{
    /* Transposes the left column to the top row for later consumption
     * by the idct/recon stage
     */
    unsigned char *left = predict - 1;
    unsigned char *above = predict - stride;
    unsigned char  p = above[-1];
    int            i, j;

    for (j = 0; j < n; j++)
    {
        for (i = 0; i < n; i++)
            predict[i] = CLAMP_255(*left + above[i] - p);

        predict += stride;
        left += stride;
    }
}

static void
predict_dc_nxn(unsigned char *predict,
               int           stride,
               int           n)
{
    unsigned char *left = predict - 1;
    unsigned char *above = predict - stride;
    int            i, j, dc = 0;

    for (i = 0; i < n; i++)
    {
        dc += *left + above[i];
        left += stride;
    }

    switch (n)
    {
    case 16:
        dc = (dc + 16) >> 5;
        break;

```

case 8:

```

        dc = (dc + 8) >> 4;
        break;
    case 4:
        dc = (dc + 4) >> 3;
        break;
    }

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            predict[i * stride + j] = dc;
}

```

```

static void
predict_ve_4x4(unsigned char *predict,
               int stride)
{
    unsigned char *above = predict - stride;
    int i, j;

    predict[0] = (above[-1] + 2 * above[0] + above[1] + 2) >> 2;
    predict[1] = (above[ 0] + 2 * above[1] + above[2] + 2) >> 2;
    predict[2] = (above[ 1] + 2 * above[2] + above[3] + 2) >> 2;
    predict[3] = (above[ 2] + 2 * above[3] + above[4] + 2) >> 2;

    for (i = 1; i < 4; i++)
        for (j = 0; j < 4; j++)
            predict[i * stride + j] = predict[j];
}

```

```

static void
predict_he_4x4(unsigned char *predict,
               int stride)
{
    unsigned char *left = predict - 1;

    predict[0] =
    predict[1] =
    predict[2] =
    predict[3] = (left[-stride] + 2 * left[0] + left[stride] + 2) >> 2;
    predict += stride;
    left += stride;

    predict[0] =
    predict[1] =
    predict[2] =

```

```
predict[3] = (left[-stride] + 2 * left[0] + left[stride] + 2) >> 2;
predict += stride;
left += stride;

predict[0] =
predict[1] =
predict[2] =
predict[3] = (left[-stride] + 2 * left[0] + left[stride] + 2) >> 2;
predict += stride;
left += stride;

predict[0] =
predict[1] =
predict[2] =
predict[3] = (left[-stride] + 2 * left[0] + left[0] + 2) >> 2;
}
```

```
static void
predict_ld_4x4(unsigned char *predict,
               int           stride)
{
    unsigned char *above = predict - stride;
    int           pred0, pred1, pred2, pred3, pred4, pred5, pred6;

    predict[0] = pred0 = (above[0] + 2 * above[1] + above[2] + 2) >> 2;
    predict[1] = pred1 = (above[1] + 2 * above[2] + above[3] + 2) >> 2;
    predict[2] = pred2 = (above[2] + 2 * above[3] + above[4] + 2) >> 2;
    predict[3] = pred3 = (above[3] + 2 * above[4] + above[5] + 2) >> 2;
    predict += stride;

    predict[0] = pred1;
    predict[1] = pred2;
    predict[2] = pred3;
    predict[3] = pred4 = (above[4] + 2 * above[5] + above[6] + 2) >> 2;
    predict += stride;

    predict[0] = pred2;
    predict[1] = pred3;
    predict[2] = pred4;
    predict[3] = pred5 = (above[5] + 2 * above[6] + above[7] + 2) >> 2;
    predict += stride;

    predict[0] = pred3;
    predict[1] = pred4;
    predict[2] = pred5;
```

```

    predict[3] = pred6 = (above[6] + 2 * above[7] + above[7] + 2) >> 2;
}

```

```

static void
predict_rd_4x4(unsigned char *predict,
               int          stride)
{
    unsigned char *left = predict - 1;
    unsigned char *above = predict - stride;
    int          pred0, pred1, pred2, pred3, pred4, pred5, pred6;

    predict[0] = pred0 =
        (left[ 0] + 2 * above[-1] + above[0] + 2) >> 2;
    predict[1] = pred1 =
        (above[-1] + 2 * above[ 0] + above[1] + 2) >> 2;
    predict[2] = pred2 =
        (above[ 0] + 2 * above[ 1] + above[2] + 2) >> 2;
    predict[3] = pred3 =
        (above[ 1] + 2 * above[ 2] + above[3] + 2) >> 2;
    predict += stride;

    predict[0] = pred4 =
        (left[stride] + 2 * left[0] + above[-1] + 2) >> 2;
    predict[1] = pred0;
    predict[2] = pred1;
    predict[3] = pred2;
    predict += stride;

    predict[0] = pred5 =
        (left[stride*2] + 2 * left[stride] + left[0] + 2) >> 2;
    predict[1] = pred4;
    predict[2] = pred0;
    predict[3] = pred1;
    predict += stride;

    predict[0] = pred6 =
        (left[stride*3] + 2 * left[stride*2] + left[stride] + 2) >> 2;
    predict[1] = pred5;
    predict[2] = pred4;
    predict[3] = pred0;
}

```

```

static void
predict_vr_4x4(unsigned char *predict,
               int           stride)
{
    unsigned char *left = predict - 1;
    unsigned char *above = predict - stride;
    int           pred0, pred1, pred2, pred3, pred4, pred5, pred6,
               pred7, pred8, pred9;

```

```

    predict[0] = pred0 = (above[-1] + above[0] + 1) >> 1;
    predict[1] = pred1 = (above[ 0] + above[1] + 1) >> 1;
    predict[2] = pred2 = (above[ 1] + above[2] + 1) >> 1;
    predict[3] = pred3 = (above[ 2] + above[3] + 1) >> 1;
    predict += stride;

    predict[0] = pred4 =
        (left[ 0] + 2 * above[-1] + above[0] + 2) >> 2;
    predict[1] = pred5 =
        (above[-1] + 2 * above[ 0] + above[1] + 2) >> 2;
    predict[2] = pred6 =
        (above[ 0] + 2 * above[ 1] + above[2] + 2) >> 2;
    predict[3] = pred7 =
        (above[ 1] + 2 * above[ 2] + above[3] + 2) >> 2;
    predict += stride;

    predict[0] = pred8 =
        (left[stride] + 2 * left[0] + above[-1] + 2) >> 2;
    predict[1] = pred0;
    predict[2] = pred1;
    predict[3] = pred2;
    predict += stride;

    predict[0] = pred9 =
        (left[stride*2] + 2 * left[stride] + left[0] + 2) >> 2;
    predict[1] = pred4;
    predict[2] = pred5;
    predict[3] = pred6;
}

```

```

static void

```

```

predict_vl_4x4(unsigned char *predict,
               int           stride)
{
    unsigned char *above = predict - stride;
    int           pred0, pred1, pred2, pred3, pred4, pred5, pred6,
               pred7, pred8, pred9;

    predict[0] = pred0 = (above[0] + above[1] + 1) >> 1;
    predict[1] = pred1 = (above[1] + above[2] + 1) >> 1;
    predict[2] = pred2 = (above[2] + above[3] + 1) >> 1;
    predict[3] = pred3 = (above[3] + above[4] + 1) >> 1;
    predict += stride;

    predict[0] = pred4 =
        (above[0] + 2 * above[1] + above[2] + 2) >> 2;
    predict[1] = pred5 =

```

```

    (above[1] + 2 * above[2] + above[3] + 2) >> 2;
    predict[2] = pred6 =
        (above[2] + 2 * above[3] + above[4] + 2) >> 2;
    predict[3] = pred7 =
        (above[3] + 2 * above[4] + above[5] + 2) >> 2;
    predict += stride;

    predict[0] = pred1;
    predict[1] = pred2;
    predict[2] = pred3;
    predict[3] = pred8 = (above[4] + 2 * above[5] + above[6] + 2) >> 2;
    predict += stride;

    predict[0] = pred5;
    predict[1] = pred6;
    predict[2] = pred7;
    predict[3] = pred9 = (above[5] + 2 * above[6] + above[7] + 2) >> 2;
}

static void
predict_hd_4x4(unsigned char *predict,
               int           stride)
{
    unsigned char *left = predict - 1;

```

```

unsigned char *above = predict - stride;
int          pred0, pred1, pred2, pred3, pred4, pred5, pred6,
            pred7, pred8, pred9;

predict[0] = pred0 =
    (left[ 0] + above[-1] + 1) >> 1;
predict[1] = pred1 =
    (left[ 0] + 2 * above[-1] + above[0] + 2) >> 2;
predict[2] = pred2 =
    (above[-1] + 2 * above[ 0] + above[1] + 2) >> 2;
predict[3] = pred3 =
    (above[ 0] + 2 * above[ 1] + above[2] + 2) >> 2;
predict += stride;

predict[0] = pred4 =
    (left[stride] + left[0] + 1) >> 1;
predict[1] = pred5 =
    (left[stride] + 2 * left[0] + above[-1] + 2) >> 2;
predict[2] = pred0;
predict[3] = pred1;
predict += stride;

predict[0] = pred6 =

```

```

    (left[stride*2] + left[stride] + 1) >> 1;
predict[1] = pred7 =
    (left[stride*2] + 2 * left[stride] + left[0] + 2) >> 2;
predict[2] = pred4;
predict[3] = pred5;
predict += stride;

predict[0] = pred8 =
    (left[stride*3] + left[stride*2] + 1) >> 1;
predict[1] = pred9 =
    (left[stride*3] + 2 * left[stride*2] + left[stride] + 2) >> 2;
predict[2] = pred6;
predict[3] = pred7;
}

```

```

static void
predict_hu_4x4(unsigned char *predict,

```

```

        int                stride)
{
    unsigned char *left = predict - 1;
    int            pred0, pred1, pred2, pred3, pred4, pred5, pred6;

    predict[0] = pred0 =
        (left[stride*0] + left[stride*1] + 1) >> 1;
    predict[1] = pred1 =
        (left[stride*0] + 2 * left[stride*1] + left[stride*2] + 2) >> 2;
    predict[2] = pred2 =
        (left[stride*1] + left[stride*2] + 1) >> 1;
    predict[3] = pred3 =
        (left[stride*1] + 2 * left[stride*2] + left[stride*3] + 2) >> 2;
    predict += stride;

    predict[0] = pred2;
    predict[1] = pred3;
    predict[2] = pred4 =
        (left[stride*2] + left[stride*3] + 1) >> 1;
    predict[3] = pred5 =
        (left[stride*2] + 2 * left[stride*3] + left[stride*3] + 2) >> 2;
    predict += stride;

    predict[0] = pred4;
    predict[1] = pred5;
    predict[2] = pred6 = left[stride*3];
    predict[3] = pred6;
    predict += stride;

    predict[0] = pred6;

```

```

    predict[1] = pred6;
    predict[2] = pred6;
    predict[3] = pred6;
}

static void
predict_h_16x16(unsigned char *predict, int stride)
{
    predict_h_nxn(predict, stride, 16);
}

```

```
static void
predict_v_16x16(unsigned char *predict, int stride)
{
    predict_v_nxn(predict, stride, 16);
}
```

```
static void
predict_tm_16x16(unsigned char *predict, int stride)
{
    predict_tm_nxn(predict, stride, 16);
}
```

```
static void
predict_h_8x8(unsigned char *predict, int stride)
{
    predict_h_nxn(predict, stride, 8);
}
```

```
static void
predict_v_8x8(unsigned char *predict, int stride)
{
    predict_v_nxn(predict, stride, 8);
}
```

```
static void
predict_tm_8x8(unsigned char *predict, int stride)
{
    predict_tm_nxn(predict, stride, 8);
}
```

```
static void
predict_tm_4x4(unsigned char *predict, int stride)
{
    predict_tm_nxn(predict, stride, 4);
}
```

```
}
```

```
static void
copy_down(unsigned char      *recon,
           int                stride)
{
    /* Copy the four pixels above-right of subblock 3 to
     * above-right of subblocks 7, 11, and 15
     */
    uint32_t tmp, *copy = (void *) (recon + 16 - stride);

    stride = stride / sizeof(unsigned int);
    tmp = *copy;
    copy += stride * 4;
    *copy = tmp;
    copy += stride * 4;
    *copy = tmp;
    copy += stride * 4;
    *copy = tmp;
}
```

```
static void
b_pred(unsigned char *predict,
        int          stride,
        struct mb_info *mbi,
        short         *coeffs)
{
    int i;

    copy_down(predict, stride);

    for (i = 0; i < 16; i++)
    {
        unsigned char *b_predict = predict + (i & 3) * 4;

        switch (mbi->split.modes[i])
        {
            case B_DC_PRED:
                predict_dc_nxn(b_predict, stride, 4);
                break;
            case B_TM_PRED:
                predict_tm_4x4(b_predict, stride);
        }
    }
}
```



```
        break;
    case B_VE_PRED:
        predict_ve_4x4(b_predict, stride);
        break;
    case B_HE_PRED:
        predict_he_4x4(b_predict, stride);
        break;
    case B_LD_PRED:
        predict_ld_4x4(b_predict, stride);
        break;
    case B_RD_PRED:
        predict_rd_4x4(b_predict, stride);
        break;
    case B_VR_PRED:
        predict_vr_4x4(b_predict, stride);
        break;
    case B_VL_PRED:
        predict_vl_4x4(b_predict, stride);
        break;
    case B_HD_PRED:
        predict_hd_4x4(b_predict, stride);
        break;
    case B_HU_PRED:
        predict_hu_4x4(b_predict, stride);
        break;
    default:
        assert(0);
}

vp8_dixie_idct_add(b_predict, b_predict, stride, coeffs);
coeffs += 16;

if ((i & 3) == 3)
{
    predict += stride * 4;
}
}

static void
fixup_dc_coeffs(struct mb_info *mbi,
                short          *coeffs)
{
    short y2[16];
    int   i;
```

```
vp8_dixie_walsh(coeffs + 24 * 16, y2);
```

```
    for (i = 0; i < 16; i++)
        coeffs[i*16] = y2[i];
}

static void
predict_intra_luma(unsigned char *predict,
                  int stride,
                  struct mb_info *mbi,
                  short *coeffs)
{
    if (mbi->base.y_mode == B_PRED)
        b_pred(predict, stride, mbi, coeffs);
    else
    {
        int i;

        switch (mbi->base.y_mode)
        {
            case DC_PRED:
                predict_dc_nxn(predict, stride, 16);
                break;
            case V_PRED:
                predict_v_16x16(predict, stride);
                break;
            case H_PRED:
                predict_h_16x16(predict, stride);
                break;
            case TM_PRED:
                predict_tm_16x16(predict, stride);
                break;
            default:
                assert(0);
        }

        fixup_dc_coeffs(mbi, coeffs);

        for (i = 0; i < 16; i++)
        {
            vp8_dixie_idct_add(predict, predict, stride, coeffs);
        }
    }
}
```

```

        coeffs += 16;
        predict += 4;

        if ((i & 3) == 3)
            predict += stride * 4 - 16;
    }

}

```

```

}

```

```

static void
predict_intra_chroma(unsigned char *predict_u,
                    unsigned char *predict_v,
                    int stride,
                    struct mb_info *mbi,
                    short *coeffs)
{
    int i;

    switch (mbi->base.uv_mode)
    {
    case DC_PRED:
        predict_dc_nxn(predict_u, stride, 8);
        predict_dc_nxn(predict_v, stride, 8);
        break;
    case V_PRED:
        predict_v_8x8(predict_u, stride);
        predict_v_8x8(predict_v, stride);
        break;
    case H_PRED:
        predict_h_8x8(predict_u, stride);
        predict_h_8x8(predict_v, stride);
        break;
    case TM_PRED:
        predict_tm_8x8(predict_u, stride);
        predict_tm_8x8(predict_v, stride);
        break;
    default:
        assert(0);
    }
}

```

```

coeffs += 16 * 16;

for (i = 16; i < 20; i++)
{
    vp8_dixie_idct_add(predict_u, predict_u, stride, coeffs);
    coeffs += 16;
    predict_u += 4;

    if (i & 1)
        predict_u += stride * 4 - 8;
}

for (i = 20; i < 24; i++)
{

```

```

    vp8_dixie_idct_add(predict_v, predict_v, stride, coeffs);
    coeffs += 16;
    predict_v += 4;

    if (i & 1)
        predict_v += stride * 4 - 8;
}

}

```

```

static void
sixtap_horiz(unsigned char      *output,
              int                output_stride,
              const unsigned char *reference,
              int                reference_stride,
              int                cols,
              int                rows,
              const filter_t      filter
              )
{
    int r, c, temp;

    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < cols; c++)

```

```

    {
        temp = (reference[-2] * filter[0]) +
               (reference[-1] * filter[1]) +
               (reference[ 0] * filter[2]) +
               (reference[ 1] * filter[3]) +
               (reference[ 2] * filter[4]) +
               (reference[ 3] * filter[5]) +
               64;
        temp >>= 7;
        output[c] = CLAMP_255(temp);
        reference++;
    }

    reference += reference_stride - cols;
    output += output_stride;
}

```

```

static void
sixtap_vert(unsigned char      *output,
             int               output_stride,

```

```

        const unsigned char *reference,
        int                 reference_stride,
        int                 cols,
        int                 rows,
        const filter_t      filter
    )
{
    int r, c, temp;

    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < cols; c++)
        {
            temp = (reference[-2*reference_stride] * filter[0]) +
                   (reference[-1*reference_stride] * filter[1]) +
                   (reference[ 0*reference_stride] * filter[2]) +
                   (reference[ 1*reference_stride] * filter[3]) +
                   (reference[ 2*reference_stride] * filter[4]) +
                   (reference[ 3*reference_stride] * filter[5]) +

```

```

        64;
        temp >>= 7;
        output[c] = CLAMP_255(temp);
        reference++;
    }

    reference += reference_stride - cols;
    output += output_stride;
}
}

```

```

static void
sixtap_2d(unsigned char      *output,
          int                output_stride,
          const unsigned char *reference,
          int                reference_stride,
          int                cols,
          int                rows,
          int                mx,
          int                my,
          const filter_t     filters[8])
{
    DECLARE_ALIGNED(16, unsigned char, temp[16*(16+5)]);

    sixtap_horiz(temp, 16,
                 reference - 2 * reference_stride, reference_stride,
                 cols, rows + 5, filters[mx]);
}

```

Bankoski, et al. Expires November 19, 2011 [Page 223]

Internet-Draft VP8 Data Format and Decoding Guide May 2011

```

    sixtap_vert(output, output_stride,
                 temp + 2 * 16, 16,
                 cols, rows, filters[my]);
}

```

```

struct img_index
{
    unsigned char *y, *u, *v;
    int          stride, uv_stride;
};

```

```

static const unsigned char *
filter_block(unsigned char      *output,
              const unsigned char *reference,
              int                stride,
              const union mv     *mv,
              const filter_t     filters[8])
{
    int mx, my;

    /* Handle 0,0 as a special case. TODO: does this make it any
     * faster?
     */
    if (!mv->raw)
        return reference;

    mx = mv->d.x & 7;
    my = mv->d.y & 7;
    reference += ((mv->d.y >> 3) * stride) + (mv->d.x >> 3);

    if (mx | my)
    {
        sixtap_2d(output, stride, reference, stride, 4, 4, mx, my,
                  filters);
        reference = output;
    }

    return reference;
}

```

```

static void
recon_1_block(unsigned char      *output,
               const unsigned char *reference,
               int                stride,
               const union mv     *mv,

```

```

    const filter_t     filters[8],
    short               *coeffs,
    struct mb_info      *mbi,
    int                 b
)

```

```

{
    const unsigned char *predict;

    predict = filter_block(output, reference, stride, mv, filters);
    vp8_dixie_idct_add(output, predict, stride, coeffs + 16 * b);
}

```

```

static mv_t
calculate_chroma_splitmv(struct mb_info *mbi,
                        int          b,
                        int          full_pixel)
{
    int temp;
    union mv mv;

    temp = mbi->split.mvs[b].d.x +
           mbi->split.mvs[b+1].d.x +
           mbi->split.mvs[b+4].d.x +
           mbi->split.mvs[b+5].d.x;

    if (temp < 0)
        temp -= 4;
    else
        temp += 4;

    mv.d.x = temp / 8;

    temp = mbi->split.mvs[b].d.y +
           mbi->split.mvs[b+1].d.y +
           mbi->split.mvs[b+4].d.y +
           mbi->split.mvs[b+5].d.y;

    if (temp < 0)
        temp -= 4;
    else
        temp += 4;

    mv.d.y = temp / 8;

    if (full_pixel)
    {
        mv.d.x &= ~7;
    }
}

```



```

        mv.d.y &= ~7;
    }

    return mv;
}

```

```

/* Note: We rely on the reconstructed border having the same stride as
 * the reference buffer because the filter_block can't adjust the
 * stride with its return value, only the reference pointer.
 */

```

```

static void

```

```

build_mc_border(unsigned char      *dst,
                const unsigned char *src,
                int                 stride,
                int                 x,
                int                 y,
                int                 b_w,
                int                 b_h,
                int                 w,
                int                 h
                )

```

```

{
    const unsigned char *ref_row;

```

```

    /* Get a pointer to the start of the real data for this row */
    ref_row = src - x - y * stride;

```

```

    if (y >= h)
        ref_row += (h - 1) * stride;
    else if (y > 0)
        ref_row += y * stride;

```

```

    do

```

```

    {
        int left, right = 0, copy;

```

```

        left = x < 0 ? -x : 0;

```

```

        if (left > b_w)
            left = b_w;

```

```

        if (x + b_w > w)
            right = x + b_w - w;

```

```

        if (right > b_w)
            right = b_w;

```

```
    copy = b_w - left - right;

    if (left)
        memset(dst, ref_row[0], left);

    if (copy)
        memcpy(dst + left, ref_row + x + left, copy);

    if (right)
        memset(dst + left + copy, ref_row[w-1], right);

    dst += stride;
    y++;

    if (y < h && y > 0)
        ref_row += stride;
}
while (--b_h);
}
```

```
static void
recon_1_edge_block(unsigned char          *output,
                  unsigned char          *emul_block,
                  const unsigned char    *reference,
                  int                    stride,
                  const union mv         *mv,
                  const filter_t         filters[8],
                  short                  *coeffs,
                  struct mb_info         *mbi,
                  int                    x,
                  int                    y,
                  int                    w,
                  int                    h,
                  int                    start_b
                  )
{
    const unsigned char *predict;
    int                 b = start_b;
    const int           b_w = 4;
    const int           b_h = 4;
```

```

x += mv->d.x >> 3;
y += mv->d.y >> 3;

/* Need two pixels left/above, 3 right/below for 6-tap */
if (x < 2 || x + b_w - 1 + 3 >= w || y < 2 || y + b_h - 1 + 3 >= h)
{

```

```

        reference += (mv->d.x >> 3) + (mv->d.y >> 3) * stride;
        build_mc_border(emul_block,
                        reference - 2 - 2 * stride, stride,
                        x - 2, y - 2, b_w + 5, b_h + 5, w, h);
        reference = emul_block + 2 * stride + 2;
        reference -= (mv->d.x >> 3) + (mv->d.y >> 3) * stride;
    }

    predict = filter_block(output, reference, stride, mv, filters);
    vp8_dixie_idct_add(output, predict, stride, coeffs + 16 * b);
}

static void
predict_inter_emulated_edge(struct vp8_decoder_ctx *ctx,
                           struct img_index *img,
                           short *coeffs,
                           struct mb_info *mbi,
                           int mb_col,
                           int mb_row)
{
    /* TODO: move this into its own buffer. This only works because we
     * still have a border allocated.
     */
    unsigned char *emul_block = ctx->frame_strg[0].img.img_data;
    unsigned char *reference;
    unsigned char *output;
    ptrdiff_t reference_offset;
    int w, h, x, y, b;
    union mv chroma_mv[4];
    unsigned char *u = img->u, *v = img->v;
    int full_pixel = ctx->frame_hdr.version == 3;

    x = mb_col * 16;

```

```

y = mb_row * 16;
w = ctx->mb_cols * 16;
h = ctx->mb_rows * 16;
output = img->y;
reference_offset = ctx->ref_frame_offsets[mbi->base.ref_frame];
reference = output + reference_offset;

if (mbi->base.y_mode != SPLITMV)
{
    union mv uvmv;

    uvmv = mbi->base.mv;
    uvmv.d.x = (uvmv.d.x + 1 + (uvmv.d.x >> 31) * 2) / 2;

```

```

    uvmv.d.y = (uvmv.d.y + 1 + (uvmv.d.y >> 31) * 2) / 2;

    if (full_pixel)
    {
        uvmv.d.x &= ~7;
        uvmv.d.y &= ~7;
    }

    chroma_mv[0] = uvmv;
    chroma_mv[1] = uvmv;
    chroma_mv[2] = uvmv;
    chroma_mv[3] = uvmv;
}
else
{
    chroma_mv[0] = calculate_chroma_splitmv(mbi, 0, full_pixel);
    chroma_mv[1] = calculate_chroma_splitmv(mbi, 2, full_pixel);
    chroma_mv[2] = calculate_chroma_splitmv(mbi, 8, full_pixel);
    chroma_mv[3] = calculate_chroma_splitmv(mbi, 10, full_pixel);
}

/* Luma */
for (b = 0; b < 16; b++)
{
    union mv *ymv;

    if (mbi->base.y_mode != SPLITMV)

```

```

        ymv = &mbi->base.mv;
    else
        ymv = mbi->split.mvs + b;

    recon_1_edge_block(output, emul_block, reference, img->stride,
                      ymv, ctx->subpixel_filters,
                      coeffs, mbi, x, y, w, h, b);

    x += 4;
    output += 4;
    reference += 4;

    if ((b & 3) == 3)
    {
        x -= 16;
        y += 4;
        output += 4 * img->stride - 16;
        reference += 4 * img->stride - 16;
    }
}

```

```

x = mb_col * 16;
y = mb_row * 16;

/* Chroma */
x >>= 1;
y >>= 1;
w >>= 1;
h >>= 1;

for (b = 0; b < 4; b++)
{
    recon_1_edge_block(u, emul_block, u + reference_offset,
                      img->uv_stride,
                      &chroma_mv[b], ctx->subpixel_filters,
                      coeffs, mbi, x, y, w, h, b + 16);
    recon_1_edge_block(v, emul_block, v + reference_offset,
                      img->uv_stride,
                      &chroma_mv[b], ctx->subpixel_filters,
                      coeffs, mbi, x, y, w, h, b + 20);

    u += 4;
    v += 4;
}

```

```

        x += 4;

        if (b & 1)
        {
            x -= 8;
            y += 4;
            u += 4 * img->uv_stride - 8;
            v += 4 * img->uv_stride - 8;
        }
    }
}

static void
predict_inter(struct vp8_decoder_ctx *ctx,
              struct img_index *img,
              short *coeffs,
              struct mb_info *mbi)
{
    unsigned char *y = img->y;
    unsigned char *u = img->u;
    unsigned char *v = img->v;
    ptrdiff_t reference_offset;
    union mv chroma_mv[4];
    int full_pixel = ctx->frame_hdr.version == 3;
    int b;

```

```

    if (mbi->base.y_mode != SPLITMV)
    {
        union mv uvmv;

        uvmv = mbi->base.mv;
        uvmv.d.x = (uvmv.d.x + 1 + (uvmv.d.x >> 31) * 2) / 2;
        uvmv.d.y = (uvmv.d.y + 1 + (uvmv.d.y >> 31) * 2) / 2;

        if (full_pixel)
        {
            uvmv.d.x &= ~7;
            uvmv.d.y &= ~7;
        }
    }

```

```

        chroma_mv[0] =
            chroma_mv[1] =
                chroma_mv[2] =
                    chroma_mv[3] = uvmv;
    }
    else
    {
        chroma_mv[0] = calculate_chroma_splitmv(mbi, 0, full_pixel);
        chroma_mv[1] = calculate_chroma_splitmv(mbi, 2, full_pixel);
        chroma_mv[2] = calculate_chroma_splitmv(mbi, 8, full_pixel);
        chroma_mv[3] = calculate_chroma_splitmv(mbi, 10, full_pixel);
    }

    reference_offset = ctx->ref_frame_offsets[mbi->base.ref_frame];

    for (b = 0; b < 16; b++)
    {
        union mv *ymv;

        if (mbi->base.y_mode != SPLITMV)
            ymv = &mbi->base.mv;
        else
            ymv = mbi->split.mvs + b;

        recon_1_block(y, y + reference_offset, img->stride,
                      ymv, ctx->subpixel_filters, coeffs, mbi, b);
        y += 4;

        if ((b & 3) == 3)
            y += 4 * img->stride - 16;
    }

    for (b = 0; b < 4; b++)

```

```

{
    recon_1_block(u, u + reference_offset,
                  img->uv_stride, &chroma_mv[b],
                  ctx->subpixel_filters, coeffs, mbi, b + 16);
    recon_1_block(v, v + reference_offset,
                  img->uv_stride, &chroma_mv[b],
                  ctx->subpixel_filters, coeffs, mbi, b + 20);
}

```

```

        u += 4;
        v += 4;

        if (b & 1)
        {
            u += 4 * img->uv_stride - 8;
            v += 4 * img->uv_stride - 8;
        }
    }
}

void
vp8_dixie_release_ref_frame(struct ref_cnt_img *rcimg)
{
    if (rcimg)
    {
        assert(rcimg->ref_cnt);
        rcimg->ref_cnt--;
    }
}

struct ref_cnt_img *
vp8_dixie_ref_frame(struct ref_cnt_img *rcimg)
{
    rcimg->ref_cnt++;
    return rcimg;
}

struct ref_cnt_img *
vp8_dixie_find_free_ref_frame(struct ref_cnt_img *frames)
{
    int i;

    for (i = 0; i < NUM_REF_FRAMES; i++)
        if (frames[i].ref_cnt == 0)
        {
            frames[i].ref_cnt = 1;
            return &frames[i];
        }
}

```



```

    }

    assert(0);
    return NULL;
}

static void
fixup_left(unsigned char      *predict,
           int                width,
           int                stride,
           unsigned int       row,
           enum prediction_mode mode)
{
    /* The left column of out-of-frame pixels is taken to be 129,
     * unless we're doing DC_PRED, in which case we duplicate the
     * above row, unless this is also row 0, in which case we use
     * 129.
     */
    unsigned char *left = predict - 1;
    int i;

    if (mode == DC_PRED && row)
    {
        unsigned char *above = predict - stride;

        for (i = 0; i < width; i++)
        {
            *left = above[i];
            left += stride;
        }
    }
    else
    {
        /* Need to re-set the above row, in case the above MB was
         * DC_PRED.
         */
        left -= stride;

        for (i = -1; i < width; i++)
        {
            *left = 129;
            left += stride;
        }
    }
}

```

```
static void
fixup_above(unsigned char      *predict,
            int                width,
            int                stride,
            unsigned int       col,
            enum prediction_mode mode)
{
    /* The above row of out-of-frame pixels is taken to be 127,
     * unless we're doing DC_PRED, in which case we duplicate the
     * left col, unless this is also col 0, in which case we use
     * 127.
     */
    unsigned char *above = predict - stride;
    int i;

    if (mode == DC_PRED && col)
    {
        unsigned char *left = predict - 1;

        for (i = 0; i < width; i++)
        {
            above[i] = *left;
            left += stride;
        }
    }
    else
    {
        /* Need to re-set the left col, in case the last MB was
         * DC_PRED.
         */
        memset(above - 1, 127, width + 1);

        memset(above + width, 127, 4); // for above-right subblock modes
    }
}

void
vp8_dixie_predict_init(struct vp8_decoder_ctx *ctx)
{
    int i;
    unsigned char *this_frame_base;

    if (ctx->frame_hdr.frame_size_updated)
```

```

{
    for (i = 0; i < NUM_REF_FRAMES; i++)
    {
        unsigned int w = ctx->mb_cols * 16 + BORDER_PIXELS * 2;
        unsigned int h = ctx->mb_rows * 16 + BORDER_PIXELS * 2;

```

```

    vpx_img_free(&ctx->frame_strg[i].img);
    ctx->frame_strg[i].ref_cnt = 0;
    ctx->ref_frames[i] = NULL;

    if (!vpx_img_alloc(&ctx->frame_strg[i].img,
                      IMG_FMT_I420, w, h, 16))
        vpx_internal_error(&ctx->error, VPX_CODEC_MEM_ERROR,
                          "Failed to allocate %dx%d"
                          " framebuffer",
                          w, h);

    vpx_img_set_rect(&ctx->frame_strg[i].img,
                     BORDER_PIXELS, BORDER_PIXELS,
                     ctx->frame_hdr.kf.w, ctx->frame_hdr.kf.h);
}

if (ctx->frame_hdr.version)
    ctx->subpixel_filters = bilinear_filters;
else
    ctx->subpixel_filters = sixtap_filters;
}

/* Find a free framebuffer to predict into */
if (ctx->ref_frames[CURRENT_FRAME])
    vp8_dixie_release_ref_frame(ctx->ref_frames[CURRENT_FRAME]);

ctx->ref_frames[CURRENT_FRAME] =
    vp8_dixie_find_free_ref_frame(ctx->frame_strg);
this_frame_base = ctx->ref_frames[CURRENT_FRAME]->img.img_data;

/* Calculate offsets to the other reference frames */
for (i = 0; i < NUM_REF_FRAMES; i++)
{
    struct ref_cnt_img *ref = ctx->ref_frames[i];

```

```

        ctx->ref_frame_offsets[i] =
            ref ? ref->img.img_data - this_frame_base : 0;
    }

    /* TODO: No need to do this on every frame... */
}

```

```

void
vp8_dixie_predict_destroy(struct vp8_decoder_ctx *ctx)
{
    int i;

```

```

    for (i = 0; i < NUM_REF_FRAMES; i++)
    {
        vpx_img_free(&ctx->frame_strg[i].img);
        ctx->frame_strg[i].ref_cnt = 0;
        ctx->ref_frames[i] = NULL;
    }
}

void
vp8_dixie_predict_process_row(struct vp8_decoder_ctx *ctx,
                             unsigned int          row,
                             unsigned int          start_col,
                             unsigned int          num_cols)
{
    struct img_index img;
    struct mb_info *mbi;
    unsigned int     col;
    short            *coeffs;

    /* Adjust pointers based on row, start_col */
    img.stride = ctx->ref_frames[CURRENT_FRAME]->img.stride[PLANE_Y];
    img.uv_stride = ctx->ref_frames[CURRENT_FRAME]->img.stride[PLANE_U];
    img.y = ctx->ref_frames[CURRENT_FRAME]->img.planes[PLANE_Y];
    img.u = ctx->ref_frames[CURRENT_FRAME]->img.planes[PLANE_U];
    img.v = ctx->ref_frames[CURRENT_FRAME]->img.planes[PLANE_V];
    img.y += (img.stride * row + start_col) * 16;
    img.u += (img.uv_stride * row + start_col) * 8;
    img.v += (img.uv_stride * row + start_col) * 8;

```



```

        predict_inter(ctx, &img, coeffs, mbi);
    }

    /* Advance to the next macroblock */
    mbi++;
    img.y += 16;
    img.u += 8;
    img.v += 8;
    coeffs += 25 * 16;
}

if (col == ctx->mb_cols)
{
    /* Extend the last row by four pixels for intra prediction.
     * This will be propagated later by copy_down.
     */
    uint32_t *extend = (uint32_t *) (img.y + 15 * img.stride);
    uint32_t val = 0x01010101 * img.y[-1 + 15 * img.stride];
    *extend = val;
}
}

```

---- End code block -----

[20.15.](#) predict.h

---- Begin code block -----

```

/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */
#ifndef PREDICT_H
#define PREDICT_H

```

```

void
vp8_dixie_predict_init(struct vp8_decoder_ctx *ctx);

void
vp8_dixie_predict_destroy(struct vp8_decoder_ctx *ctx);

void
vp8_dixie_predict_process_row(struct vp8_decoder_ctx *ctx,
                               unsigned int      row,
                               unsigned int      start_col,
                               unsigned int      num_cols);

void
vp8_dixie_release_ref_frame(struct ref_cnt_img *rcimg);

struct ref_cnt_img *
vp8_dixie_ref_frame(struct ref_cnt_img *rcimg);

struct ref_cnt_img *
vp8_dixie_find_free_ref_frame(struct ref_cnt_img *frames);

#endif

```

---- End code block -----

[20.16.](#) tokens.c

---- Begin code block -----

```

/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.

```

```

*/
#include "vpx_codec_internal.h"
#include "dixie.h"
#include "tokens.h"
#include <stdlib.h>
#include <string.h>
#include <malloc.h>

```

```

enum
{
    EOB_CONTEXT_NODE,
    ZERO_CONTEXT_NODE,
    ONE_CONTEXT_NODE,
    LOW_VAL_CONTEXT_NODE,
    TWO_CONTEXT_NODE,
    THREE_CONTEXT_NODE,
    HIGH_LOW_CONTEXT_NODE,
    CAT_ONE_CONTEXT_NODE,
    CAT_THREEFOUR_CONTEXT_NODE,
    CAT_THREE_CONTEXT_NODE,
    CAT_FIVE_CONTEXT_NODE
};
enum
{
    ZERO_TOKEN,
    ONE_TOKEN,
    TWO_TOKEN,
    THREE_TOKEN,
    FOUR_TOKEN,
    DCT_VAL_CATEGORY1,
    DCT_VAL_CATEGORY2,
    DCT_VAL_CATEGORY3,
    DCT_VAL_CATEGORY4,
    DCT_VAL_CATEGORY5,
    DCT_VAL_CATEGORY6,
    DCT_EOB_TOKEN,
    MAX_ENTROPY_TOKENS
};
struct extrabits

```

```

{

```



```

    short      min_val;
    short      length;
    unsigned char probs[12];
};
static const unsigned int left_context_index[25] =
{
    0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3,
    4, 4, 5, 5, 6, 6, 7, 7, 8
};
static const unsigned int above_context_index[25] =
{
    0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3,
    4, 5, 4, 5, 6, 7, 6, 7, 8
};
#define X(n) ((n) * PREV_COEF_CONTEXTS * ENTROPY_NODES)
static const unsigned int bands_x[16] =
{
    X(0), X(1), X(2), X(3), X(6), X(4), X(5), X(6),
    X(6), X(6), X(6), X(6), X(6), X(6), X(6), X(7)
};
#undef X
static const struct extrabits extrabits[MAX_ENTROPY_TOKENS] =
{
    { 0, -1, { 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //ZERO_TOKEN
    { 1, 0, { 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //ONE_TOKEN
    { 2, 0, { 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //TWO_TOKEN
    { 3, 0, { 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //THREE_TOKEN
    { 4, 0, { 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //FOUR_TOKEN
    { 5, 0, { 159, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //DCT_VAL_CATEGORY1
    { 7, 1, { 145, 165, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //DCT_VAL_CATEGORY2
    { 11, 2, { 140, 148, 173, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //DCT_VAL_CATEGORY3
    { 19, 3, { 135, 140, 155, 176, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //DCT_VAL_CATEGORY4
    { 35, 4, { 130, 134, 141, 157, 180, 0,
               0, 0, 0, 0, 0, 0 } }, //DCT_VAL_CATEGORY5
    { 67, 10, { 129, 130, 133, 140, 153, 177,
                196, 230, 243, 254, 254, 0 } }, //DCT_VAL_CATEGORY6
    { 0, -1, { 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0 } }, //EOB_TOKEN

```

```
};
static const unsigned int zigzag[16] =
{
    0, 1, 4, 8, 5, 2, 3, 6, 9, 12, 13, 10, 7, 11, 14, 15
};

#define DECODE_AND_APPLYSIGN(value_to_sign) \
    v = (bool_get_bit(bool) ? -value_to_sign \
        : value_to_sign) * dqf[!!c];

#define DECODE_AND_BRANCH_IF_ZERO(probability,branch) \
    if (!bool_get(bool, probability)) goto branch;

#define DECODE_AND_LOOP_IF_ZERO(probability,branch) \
    if (!bool_get(bool, probability)) \
    { \
        prob = type_probs; \
        if(c<15) {\
            ++c; \
            prob += bands_x[c]; \
            goto branch; \
        }\
        else \
            goto BLOCK_FINISHED; /*for malformed input */\
    }

#define DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val) \
    DECODE_AND_APPLYSIGN(val) \
    prob = type_probs + (ENTROPY_NODES*2); \
    if(c < 15){\
        b_tokens[zigzag[c]] = v; \
        ++c; \
        goto DO_WHILE; }\
    b_tokens[zigzag[15]] = v; \
    goto BLOCK_FINISHED;

#define DECODE_EXTRABIT_AND_ADJUST_VAL(t,bits_count)\
    val += bool_get(bool, extrabits[t].probs[bits_count]) << bits_count;

static int
decode_mb_tokens(struct bool_decoder *bool,
                token_entropy_ctx_t left,
                token_entropy_ctx_t above,
                short *tokens,
```

```
enum prediction_mode mode,  
coeff_probs_table_t probs,
```

```
        short                factor[TOKEN_BLOCK_TYPES][2])  
{  
    int            i, stop, type;  
    int            c, t, v;  
    int            val, bits_count;  
    int            eob_mask;  
    short          *b_tokens; /* tokens for this block */  
    unsigned char *type_probs; /* probabilities for this block type */  
    unsigned char *prob;  
    short          *dqf;  
  
    eob_mask = 0;  
  
    if (mode != B_PRED && mode != SPLITMV)  
    {  
        i = 24;  
        stop = 24;  
        type = 1;  
        b_tokens = tokens + 24 * 16;  
        dqf = factor[TOKEN_BLOCK_Y2];  
    }  
    else  
    {  
        i = 0;  
        stop = 16;  
        type = 3;  
        b_tokens = tokens;  
        dqf = factor[TOKEN_BLOCK_Y1];  
    }  
  
    /* Save a pointer to the coefficient probs for the current type.  
     * Need to repeat this whenever type changes.  
     */  
    type_probs = probs[type][0][0];  
  
BLOCK_LOOP:  
    t = left[left_context_index[i]] + above[above_context_index[i]];  
    c = !type; /* all blocks start at 0 except type 0, which starts  
               * at 1. */
```

```

    prob = type_probs;
    prob += t * ENTROPY_NODES;

DO_WHILE:
    prob += bands_x[c];
    DECODE_AND_BRANCH_IF_ZERO(prob[EOB_CONTEXT_NODE], BLOCK_FINISHED);

CHECK_0_:

```

```

    DECODE_AND_LOOP_IF_ZERO(prob[ZERO_CONTEXT_NODE], CHECK_0_);
    DECODE_AND_BRANCH_IF_ZERO(prob[ONE_CONTEXT_NODE],
                                ONE_CONTEXT_NODE_0_);
    DECODE_AND_BRANCH_IF_ZERO(prob[LOW_VAL_CONTEXT_NODE],
                                LOW_VAL_CONTEXT_NODE_0_);
    DECODE_AND_BRANCH_IF_ZERO(prob[HIGH_LOW_CONTEXT_NODE],
                                HIGH_LOW_CONTEXT_NODE_0_);
    DECODE_AND_BRANCH_IF_ZERO(prob[CAT_THREEFOUR_CONTEXT_NODE],
                                CAT_THREEFOUR_CONTEXT_NODE_0_);
    DECODE_AND_BRANCH_IF_ZERO(prob[CAT_FIVE_CONTEXT_NODE],
                                CAT_FIVE_CONTEXT_NODE_0_);
    val = extrabits[DCT_VAL_CATEGORY6].min_val;
    bits_count = extrabits[DCT_VAL_CATEGORY6].length;

    do
    {
        DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY6, bits_count);
        bits_count -- ;
    }
    while (bits_count >= 0);

    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

CAT_FIVE_CONTEXT_NODE_0_:
    val = extrabits[DCT_VAL_CATEGORY5].min_val;
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 4);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 3);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 2);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 1);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 0);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

```

```

CAT_THREEFOUR_CONTEXT_NODE_0_:
    DECODE_AND_BRANCH_IF_ZERO(prob[CAT_THREE_CONTEXT_NODE],
                               CAT_THREE_CONTEXT_NODE_0_);
    val = extrabits[DCT_VAL_CATEGORY4].min_val;
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY4, 3);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY4, 2);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY4, 1);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY4, 0);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

```

```

CAT_THREE_CONTEXT_NODE_0_:
    val = extrabits[DCT_VAL_CATEGORY3].min_val;
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY3, 2);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY3, 1);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY3, 0);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

```

```

HIGH_LOW_CONTEXT_NODE_0_:
    DECODE_AND_BRANCH_IF_ZERO(prob[CAT_ONE_CONTEXT_NODE],
                               CAT_ONE_CONTEXT_NODE_0_);

    val = extrabits[DCT_VAL_CATEGORY2].min_val;
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY2, 1);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY2, 0);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

```

```

CAT_ONE_CONTEXT_NODE_0_:
    val = extrabits[DCT_VAL_CATEGORY1].min_val;
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY1, 0);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

```

```

LOW_VAL_CONTEXT_NODE_0_:
    DECODE_AND_BRANCH_IF_ZERO(prob[TWO_CONTEXT_NODE],
                               TWO_CONTEXT_NODE_0_);
    DECODE_AND_BRANCH_IF_ZERO(prob[THREE_CONTEXT_NODE],
                               THREE_CONTEXT_NODE_0_);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(4);

```

```

THREE_CONTEXT_NODE_0_:
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(3);

```

```

TWO_CONTEXT_NODE_0_:

```

```

        DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(2);

ONE_CONTEXT_NODE_0_:
    DECODE_AND_APPLYSIGN(1);
    prob = type_probs + ENTROPY_NODES;

    if (c < 15)
    {
        b_tokens[zigzag[c]] = v;
        ++c;
        goto DO_WHILE;
    }

    b_tokens[zigzag[15]] = v;
BLOCK_FINISHED:
    eob_mask |= (c > 1) << i;
    t = (c != !type);    // any nonzero data?
    eob_mask |= t << 31;

    left[left_context_index[i]] = above[above_context_index[i]] = t;
    b_tokens += 16;

    i++;

```

```

    if (i < stop)
        goto BLOCK_LOOP;

    if (i == 25)
    {
        type = 0;
        i = 0;
        stop = 16;
        type_probs = probs[type][0][0];
        b_tokens = tokens;
        dqf = factor[TOKEN_BLOCK_Y1];
        goto BLOCK_LOOP;
    }

    if (i == 16)
    {
        type = 2;
        type_probs = probs[type][0][0];

```

```

        stop = 24;
        dqf = factor[TOKEN_BLOCK_UV];
        goto BLOCK_LOOP;
    }

    return eob_mask;
}

static void
reset_row_context(token_entropy_ctx_t *left)
{
    memset(left, 0, sizeof(*left));
}

static void
reset_above_context(token_entropy_ctx_t *above, unsigned int cols)
{
    memset(above, 0, cols * sizeof(*above));
}

static void
reset_mb_context(token_entropy_ctx_t *left,
                 token_entropy_ctx_t *above,
                 enum prediction_mode mode)
{
    /* Reset the macroblock context on the left and right. We have to
     * preserve the context of the second order block if this mode

```

```

    * would not have updated it.
    */
    memset(left, 0, sizeof((*left)[0]) * 8);
    memset(above, 0, sizeof((*above)[0]) * 8);

    if (mode != B_PRED && mode != SPLITMV)
    {
        (*left)[8] = 0;
        (*above)[8] = 0;
    }
}

```

```

void
vp8_dixie_tokens_process_row(struct vp8_decoder_ctx *ctx,
                           unsigned int      partition,
                           unsigned int      row,
                           unsigned int      start_col,
                           unsigned int      num_cols)
{
    struct token_decoder *tokens = &ctx->tokens[partition];
    short                *coeffs = tokens->coeffs + 25 * 16 * start_col;
    unsigned int         col;
    token_entropy_ctx_t  *above = ctx->above_token_entropy_ctx
                                + start_col;
    token_entropy_ctx_t  *left = &tokens->left_token_entropy_ctx;
    struct mb_info       *mbi = ctx->mb_info_rows[row] + start_col;

    if (row == 0)
        reset_above_context(above, num_cols);

    if (start_col == 0)
        reset_row_context(left);

    for (col = start_col; col < start_col + num_cols; col++)
    {
        memset(coeffs, 0, 25 * 16 * sizeof(short));

        if (mbi->base.skip_coeff)
        {
            reset_mb_context(left, above, mbi->base.y_mode);
            mbi->base.eob_mask = 0;
        }
        else
        {
            struct dequant_factors *dqf;

            dqf = ctx->dequant_factors + mbi->base.segment_id;

```

```

        mbi->base.eob_mask =
            decode_mb_tokens(&tokens->bool,
                           *left, *above,
                           coeffs,

```



```

        mbi->base.y_mode,
        ctx->entropy_hdr.coeff_probs,
        dqf->factor);
    }

    above++;
    mbi++;
    coeffs += 25 * 16;
}
}

void
vp8_dixie_tokens_init(struct vp8_decoder_ctx *ctx)
{
    unsigned int partitions = ctx->token_hdr.partitions;

    if (ctx->frame_hdr.frame_size_updated)
    {
        unsigned int i;
        unsigned int coeff_row_sz =
            ctx->mb_cols * 25 * 16 * sizeof(short);

        for (i = 0; i < partitions; i++)
        {
            free(ctx->tokens[i].coeffs);
            ctx->tokens[i].coeffs = memalign(16, coeff_row_sz);

            if (!ctx->tokens[i].coeffs)
                vpx_internal_error(&ctx->error, VPX_CODEC_MEM_ERROR,
                                   NULL);
        }

        free(ctx->above_token_entropy_ctx);
        ctx->above_token_entropy_ctx =
            calloc(ctx->mb_cols, sizeof(*ctx->above_token_entropy_ctx));

        if (!ctx->above_token_entropy_ctx)
            vpx_internal_error(&ctx->error, VPX_CODEC_MEM_ERROR, NULL);
    }
}

void

```

```
vp8_dixie_tokens_destroy(struct vp8_decoder_ctx *ctx)
{
    int i;

    for (i = 0; i < MAX_PARTITIONS; i++)
        free(ctx->tokens[i].coeffs);

    free(ctx->above_token_entropy_ctx);
}
```

---- End code block -----

[20.17.](#) tokens.h

---- Begin code block -----

```
/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */
#include "vpx_codec_internal.h"
#include "dixie.h"
#include "tokens.h"
#include <stdlib.h>
#include <string.h>
#include <malloc.h>

enum
{
    EOB_CONTEXT_NODE,
    ZERO_CONTEXT_NODE,
    ONE_CONTEXT_NODE,
    LOW_VAL_CONTEXT_NODE,
    TWO_CONTEXT_NODE,
    THREE_CONTEXT_NODE,
    HIGH_LOW_CONTEXT_NODE,
    CAT_ONE_CONTEXT_NODE,
    CAT_THREEFOUR_CONTEXT_NODE,
    CAT_THREE_CONTEXT_NODE,
    CAT_FIVE_CONTEXT_NODE
}
```

```
};
```

```
enum
{
    ZERO_TOKEN,
    ONE_TOKEN,
    TWO_TOKEN,
    THREE_TOKEN,
    FOUR_TOKEN,
    DCT_VAL_CATEGORY1,
    DCT_VAL_CATEGORY2,
    DCT_VAL_CATEGORY3,
    DCT_VAL_CATEGORY4,
    DCT_VAL_CATEGORY5,
    DCT_VAL_CATEGORY6,
    DCT_EOB_TOKEN,
    MAX_ENTROPY_TOKENS
};
struct extrabits
{
    short      min_val;
    short      length;
    unsigned char probs[12];
};
static const unsigned int left_context_index[25] =
{
    0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3,
    4, 4, 5, 5, 6, 6, 7, 7, 8
};
static const unsigned int above_context_index[25] =
{
    0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3,
    4, 5, 4, 5, 6, 7, 6, 7, 8
};
#define X(n) ((n) * PREV_COEF_CONTEXTS * ENTROPY_NODES)
static const unsigned int bands_x[16] =
{
    X(0), X(1), X(2), X(3), X(6), X(4), X(5), X(6),
    X(6), X(6), X(6), X(6), X(6), X(6), X(6), X(7)
};
#undef X
static const struct extrabits extrabits[MAX_ENTROPY_TOKENS] =
```

```

{
    { 0, -1, { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } }, //ZERO_TOKEN
    { 1, 0, { 0, 0, 0, 0, 0, 0, 0, 0 } }, //ONE_TOKEN
    { 2, 0, { 0, 0, 0, 0, 0, 0, 0, 0 } }, //TWO_TOKEN
    { 3, 0, { 0, 0, 0, 0, 0, 0, 0, 0 } },

```

```

    { 4, 0, { 0, 0, 0, 0, 0, 0, 0, 0 } }, //THREE_TOKEN
    { 5, 0, { 0, 0, 0, 0, 0, 0, 0, 0 } }, //FOUR_TOKEN
    { 5, 0, { 159, 0, 0, 0, 0, 0, 0, 0 } }, //DCT_VAL_CATEGORY1
    { 7, 1, { 145, 165, 0, 0, 0, 0, 0, 0 } }, //DCT_VAL_CATEGORY2
    { 11, 2, { 140, 148, 173, 0, 0, 0, 0, 0 } }, //DCT_VAL_CATEGORY3
    { 19, 3, { 135, 140, 155, 176, 0, 0, 0, 0 } }, //DCT_VAL_CATEGORY4
    { 35, 4, { 130, 134, 141, 157, 180, 0, 0, 0 } }, //DCT_VAL_CATEGORY5
    { 67, 10, { 129, 130, 133, 140, 153, 177, 196, 230, 243, 254, 254, 0 } }, //DCT_VAL_CATEGORY6
    { 0, -1, { 0, 0, 0, 0, 0, 0, 0, 0 } }, // EOB TOKEN
};
static const unsigned int zigzag[16] =
{
    0, 1, 4, 8, 5, 2, 3, 6, 9, 12, 13, 10, 7, 11, 14, 15
};

#define DECODE_AND_APPLYSIGN(value_to_sign) \
    v = (bool_get_bit(bool) ? -value_to_sign \
        : value_to_sign) * dqf[!!c];

#define DECODE_AND_BRANCH_IF_ZERO(probability,branch) \
    if (!bool_get(bool, probability)) goto branch;

#define DECODE_AND_LOOP_IF_ZERO(probability,branch) \
    if (!bool_get(bool, probability)) \
    { \

```

```

        prob = type_probs; \
        if(c<15) {\
            ++c; \
            prob += bands_x[c]; \
            goto branch; \
        }\
        else \
            goto BLOCK_FINISHED; /*for malformed input */\
    }

```

```

#define DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val) \
    DECODE_AND_APPLYSIGN(val) \
    prob = type_probs + (ENTROPY_NODES*2); \
    if(c < 15){\
        b_tokens[zigzag[c]] = v; \
    }

```

Bankoski, et al.

Expires November 19, 2011

[Page 250]

Internet-Draft

VP8 Data Format and Decoding Guide

May 2011

```

        ++c; \
        goto DO_WHILE; }\
    b_tokens[zigzag[15]] = v; \
    goto BLOCK_FINISHED;

```

```

#define DECODE_EXTRABIT_AND_ADJUST_VAL(t, bits_count)\
    val += bool_get(bool, extrabits[t].probs[bits_count]) << bits_count;

```

```

static int
decode_mb_tokens(struct bool_decoder *bool,
                 token_entropy_ctx_t left,
                 token_entropy_ctx_t above,
                 short *tokens,
                 enum prediction_mode mode,
                 coeff_probs_table_t probs,
                 short factor[TOKEN_BLOCK_TYPES][2])
{
    int i, stop, type;
    int c, t, v;
    int val, bits_count;
    int eob_mask;
    short *b_tokens; /* tokens for this block */
    unsigned char *type_probs; /* probabilities for this block type */
    unsigned char *prob;

```

```

short          *dqf;

eob_mask = 0;

if (mode != B_PRED && mode != SPLITMV)
{
    i = 24;
    stop = 24;
    type = 1;
    b_tokens = tokens + 24 * 16;
    dqf = factor[TOKEN_BLOCK_Y2];
}
else
{
    i = 0;
    stop = 16;
    type = 3;
    b_tokens = tokens;
    dqf = factor[TOKEN_BLOCK_Y1];
}

/* Save a pointer to the coefficient probs for the current type.

```

```

    * Need to repeat this whenever type changes.
    */
    type_probs = probs[type][0][0];

BLOCK_LOOP:
    t = left[left_context_index[i]] + above[above_context_index[i]];
    c = !type; /* all blocks start at 0 except type 0, which starts
               * at 1. */

    prob = type_probs;
    prob += t * ENTROPY_NODES;

DO_WHILE:
    prob += bands_x[c];
    DECODE_AND_BRANCH_IF_ZERO(prob[EOB_CONTEXT_NODE], BLOCK_FINISHED);

CHECK_0_:
    DECODE_AND_LOOP_IF_ZERO(prob[ZERO_CONTEXT_NODE], CHECK_0_);
    DECODE_AND_BRANCH_IF_ZERO(prob[ONE_CONTEXT_NODE],

```

```

                                ONE_CONTEXT_NODE_0_);
DECODE_AND_BRANCH_IF_ZERO(prob[LOW_VAL_CONTEXT_NODE],
                            LOW_VAL_CONTEXT_NODE_0_);
DECODE_AND_BRANCH_IF_ZERO(prob[HIGH_LOW_CONTEXT_NODE],
                            HIGH_LOW_CONTEXT_NODE_0_);
DECODE_AND_BRANCH_IF_ZERO(prob[CAT_THREEFOUR_CONTEXT_NODE],
                            CAT_THREEFOUR_CONTEXT_NODE_0_);
DECODE_AND_BRANCH_IF_ZERO(prob[CAT_FIVE_CONTEXT_NODE],
                            CAT_FIVE_CONTEXT_NODE_0_);
val = extrabits[DCT_VAL_CATEGORY6].min_val;
bits_count = extrabits[DCT_VAL_CATEGORY6].length;

do
{
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY6, bits_count);
    bits_count -- ;
}
while (bits_count >= 0);

DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

CAT_FIVE_CONTEXT_NODE_0_:
val = extrabits[DCT_VAL_CATEGORY5].min_val;
DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 4);
DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 3);
DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 2);
DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 1);
DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY5, 0);
DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

```

```

CAT_THREEFOUR_CONTEXT_NODE_0_:
    DECODE_AND_BRANCH_IF_ZERO(prob[CAT_THREE_CONTEXT_NODE],
                                CAT_THREE_CONTEXT_NODE_0_);
    val = extrabits[DCT_VAL_CATEGORY4].min_val;
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY4, 3);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY4, 2);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY4, 1);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY4, 0);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

CAT_THREE_CONTEXT_NODE_0_:
    val = extrabits[DCT_VAL_CATEGORY3].min_val;

```

```

    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY3, 2);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY3, 1);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY3, 0);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

HIGH_LOW_CONTEXT_NODE_0_:
    DECODE_AND_BRANCH_IF_ZERO(prob[CAT_ONE_CONTEXT_NODE],
                               CAT_ONE_CONTEXT_NODE_0_);

    val = extrabits[DCT_VAL_CATEGORY2].min_val;
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY2, 1);
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY2, 0);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

CAT_ONE_CONTEXT_NODE_0_:
    val = extrabits[DCT_VAL_CATEGORY1].min_val;
    DECODE_EXTRABIT_AND_ADJUST_VAL(DCT_VAL_CATEGORY1, 0);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(val);

LOW_VAL_CONTEXT_NODE_0_:
    DECODE_AND_BRANCH_IF_ZERO(prob[TWO_CONTEXT_NODE],
                               TWO_CONTEXT_NODE_0_);
    DECODE_AND_BRANCH_IF_ZERO(prob[THREE_CONTEXT_NODE],
                               THREE_CONTEXT_NODE_0_);
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(4);

THREE_CONTEXT_NODE_0_:
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(3);

TWO_CONTEXT_NODE_0_:
    DECODE_SIGN_WRITE_COEFF_AND_CHECK_EXIT(2);

ONE_CONTEXT_NODE_0_:
    DECODE_AND_APPLYSIGN(1);
    prob = type_probs + ENTROPY_NODES;

```

```

if (c < 15)
{
    b_tokens[zigzag[c]] = v;
    ++c;
    goto DO_WHILE;
}

```



```

    }

    b_tokens[zigzag[15]] = v;
BLOCK_FINISHED:
    eob_mask |= (c > 1) << i;
    t = (c != !type);    // any nonzero data?
    eob_mask |= t << 31;

    left[left_context_index[i]] = above[above_context_index[i]] = t;
    b_tokens += 16;

    i++;

    if (i < stop)
        goto BLOCK_LOOP;

    if (i == 25)
    {
        type = 0;
        i = 0;
        stop = 16;
        type_probs = probs[type][0][0];
        b_tokens = tokens;
        dqf = factor[TOKEN_BLOCK_Y1];
        goto BLOCK_LOOP;
    }

    if (i == 16)
    {
        type = 2;
        type_probs = probs[type][0][0];
        stop = 24;
        dqf = factor[TOKEN_BLOCK_UV];
        goto BLOCK_LOOP;
    }

    return eob_mask;
}

static void
reset_row_context(token_entropy_ctx_t *left)
{

```

```

    memset(left, 0, sizeof(*left));
}

```

```

static void
reset_above_context(token_entropy_ctx_t *above, unsigned int cols)
{
    memset(above, 0, cols * sizeof(*above));
}

```

```

static void
reset_mb_context(token_entropy_ctx_t *left,
                 token_entropy_ctx_t *above,
                 enum prediction_mode mode)
{
    /* Reset the macroblock context on the left and right. We have to
     * preserve the context of the second order block if this mode
     * would not have updated it.
     */
    memset(left, 0, sizeof((*left)[0]) * 8);
    memset(above, 0, sizeof((*above)[0]) * 8);

    if (mode != B_PRED && mode != SPLITMV)
    {
        (*left)[8] = 0;
        (*above)[8] = 0;
    }
}

```

```

void
vp8_dixie_tokens_process_row(struct vp8_decoder_ctx *ctx,
                            unsigned int          partition,
                            unsigned int          row,
                            unsigned int          start_col,
                            unsigned int          num_cols)
{
    struct token_decoder *tokens = &ctx->tokens[partition];
    short                *coeffs = tokens->coeffs + 25 * 16 * start_col;
    unsigned int         col;
    token_entropy_ctx_t *above = ctx->above_token_entropy_ctx
                                + start_col;
    token_entropy_ctx_t *left = &tokens->left_token_entropy_ctx;
    struct mb_info       *mbi = ctx->mb_info_rows[row] + start_col;

    if (row == 0)
        reset_above_context(above, num_cols);
}

```

```
    if (start_col == 0)
        reset_row_context(left);

    for (col = start_col; col < start_col + num_cols; col++)
    {
        memset(coeffs, 0, 25 * 16 * sizeof(short));

        if (mbi->base.skip_coeff)
        {
            reset_mb_context(left, above, mbi->base.y_mode);
            mbi->base.eob_mask = 0;
        }
        else
        {
            struct dequant_factors *dqf;

            dqf = ctx->dequant_factors + mbi->base.segment_id;
            mbi->base.eob_mask =
                decode_mb_tokens(&tokens->bool,
                                *left, *above,
                                coeffs,
                                mbi->base.y_mode,
                                ctx->entropy_hdr.coeff_probs,
                                dqf->factor);
        }

        above++;
        mbi++;
        coeffs += 25 * 16;
    }
}

void
vp8_dixie_tokens_init(struct vp8_decoder_ctx *ctx)
{
    unsigned int partitions = ctx->token_hdr.partitions;

    if (ctx->frame_hdr.frame_size_updated)
    {
        unsigned int i;
        unsigned int coeff_row_sz =
```

```

        ctx->mb_cols * 25 * 16 * sizeof(short);

for (i = 0; i < partitions; i++)
{
    free(ctx->tokens[i].coeffs);
    ctx->tokens[i].coeffs = memalign(16, coeff_row_sz);
}

```

```

        if (!ctx->tokens[i].coeffs)
            vpx_internal_error(&ctx->error, VPX_CODEC_MEM_ERROR,
                               NULL);
    }

    free(ctx->above_token_entropy_ctx);
    ctx->above_token_entropy_ctx =
        calloc(ctx->mb_cols, sizeof(*ctx->above_token_entropy_ctx));

    if (!ctx->above_token_entropy_ctx)
        vpx_internal_error(&ctx->error, VPX_CODEC_MEM_ERROR, NULL);
}
}

```

```

void
vp8_dixie_tokens_destroy(struct vp8_decoder_ctx *ctx)
{
    int i;

    for (i = 0; i < MAX_PARTITIONS; i++)
        free(ctx->tokens[i].coeffs);

    free(ctx->above_token_entropy_ctx);
}

```

----- End code block -----

[20.18.](#) vp8_prob_data.h

----- Begin code block -----

```

static const
unsigned char k_coeff_entropy_update_probs[BLOCK_TYPES][COEF_BANDS]

```

[PREV_COEF_CONTEXTS]

[ENTROPY_NODES] =

```
{
    {
        {
            {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, },
},
{
{
    {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, },
    },
},
};

```

```

static const
unsigned char k_default_y_mode_probs [] =
{ 112, 86, 140, 37};

```

```

static const
unsigned char k_default_uv_mode_probs [] =
{ 162, 101, 204};

```

```

static const
unsigned char k_default_coeff_probs [BLOCK_TYPES][COEF_BANDS]
[PREV_COEF_CONTEXTS][ENTROPY_NODES] =
{
    { /* block type 0 */
        { /* coeff band 0 */
            { 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}
        },
        { /* coeff band 1 */
            { 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}
        },
        { /* coeff band 2 */
            { 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}
        },
        { /* coeff band 3 */

```

```

        { 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}
    },
    { /* coeff band 4 */
        { 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}
    },
    { /* coeff band 5 */
        { 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}
    },
    { /* coeff band 6 */
        { 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}
    },
    { /* coeff band 7 */
        { 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}
    }
},
{ /* block type 1 */
    { /* coeff band 0 */
        { 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}
    },
    { /* coeff band 1 */
        { 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}
    },
    { /* coeff band 2 */
        { 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}
    },
    { /* coeff band 3 */
        { 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}
    },
    { /* coeff band 4 */
        { 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}
    },
    { /* coeff band 5 */
        { 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}
    },
    { /* coeff band 6 */
        { 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}
    },
    { /* coeff band 7 */
        { 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}
    }
},
{ /* block type 2 */
    { /* coeff band 0 */
        { 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}
    },
    { /* coeff band 1 */
        { 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}
    },
    { /* coeff band 2 */
        { 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}
    },
    { /* coeff band 3 */
        { 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}
    },
    { /* coeff band 4 */
        { 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},
    },
    { /* coeff band 5 */
        { 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},
    },
    { /* coeff band 6 */
        { 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},
    },
    { /* coeff band 7 */
        { 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},
    }
},
{ /* block type 3 */
    { /* coeff band 0 */
        { 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},
    },
    { /* coeff band 1 */
        { 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},
    },
    { /* coeff band 2 */
        { 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},
    },
    { /* coeff band 3 */
        { 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},
    },
    { /* coeff band 4 */
        { 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}
    },
    { /* coeff band 5 */
        { 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}
    },
    { /* coeff band 6 */
        { 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}
    }
};

```

```

    },
    { /* coeff band 7 */
        { 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}
    }
}
};

```

```

static const
unsigned char k_mv_entropy_update_probs[2][MV_PROB_CNT] =
{
    {
        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
    }
};

```

```

static const

```

```

unsigned char k_default_mv_probs[2][MV_PROB_CNT] =
{
    {
        162,
        128,
        225, 146, 172, 147, 214, 39, 156,
        128, 129, 132, 75, 145, 178, 206, 239, 254, 254 /* long bits */
    },
    {
        164,
        128,
        204, 170, 119, 235, 140, 230, 228,
        128, 130, 130, 74, 148, 180, 203, 236, 254, 254
    }
};

```

---- End code block -----

Bankoski, et al. Expires November 19, 2011 [Page 265]

Internet-Draft VP8 Data Format and Decoding Guide May 2011

[20.19.](#) vpx_codec_internal.h

---- Begin code block -----

```

/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */

/*! \file decoder_impl.h
 * \brief Describes the decoder algorithm interface for algorithm
 * implementations.
 *
 * This file defines the private structures and data types that are
 * only relevant to implementing an algorithm, as opposed to using
 * it.

```

```

*
* To create a decoder algorithm class, an interface structure is put
* into the global namespace:
*
* <pre>
* my_codec.c:
*     vpx_codec_iface_t my_codec = {
*         "My Codec v1.0",
*         VPX_CODEC_ALG_ABI_VERSION,
*         ...
*     };
* </pre>
*
* An application instantiates a specific decoder instance by using
* vpx_codec_init() and a pointer to the algorithm's interface
* structure:
*
* <pre>
* my_app.c:
*     extern vpx_codec_iface_t my_codec;
*     {
*         vpx_codec_ctx_t algo;
*         res = vpx_codec_init(&algo, &my_codec);
*     }
* </pre>
*
* Once initialized, the instance is managed using other functions
* from the vpx_codec_* family.

```

```

*/
#ifndef VPX_CODEC_INTERNAL_H
#define VPX_CODEC_INTERNAL_H
#include "vpx_decoder.h"
#include <stdarg.h>

/*!\brief Current ABI version number
*
* \internal
* If this file is altered in any way that changes the ABI, this
* value must be bumped. Examples include, but are not limited to,
* changing types, removing or reassigning enums,
* adding/removing/rearranging fields to structures
*/

```

```

#define VPX_CODEC_INTERNAL_ABI_VERSION (3)

typedef struct vpx_codec_alg_priv vpx_codec_alg_priv_t;

/*!\brief init function pointer prototype
 *
 * Performs algorithm-specific initialization of the decoder context.
 * This function is called by the generic vpx_codec_init() wrapper
 * function, so plugins implementing this interface may trust the
 * input parameters to be properly initialized.
 *
 * \param[in] ctx    Pointer to this instance's context
 * \retval #VPX_CODEC_OK
 *         The input stream was recognized and decoder initialized.
 * \retval #VPX_CODEC_MEM_ERROR
 *         Memory operation failed.
 */
typedef vpx_codec_err_t (*vpx_codec_init_fn_t)(vpx_codec_ctx_t *ctx);

/*!\brief destroy function pointer prototype
 *
 * Performs algorithm-specific destruction of the decoder context.
 * This function is called by the generic vpx_codec_destroy() wrapper
 * function, so plugins implementing this interface may trust the
 * input parameters to be properly initialized.
 *
 * \param[in] ctx    Pointer to this instance's context
 * \retval #VPX_CODEC_OK
 *         The input stream was recognized and decoder initialized.
 * \retval #VPX_CODEC_MEM_ERROR
 *         Memory operation failed.
 */
typedef vpx_codec_err_t (*vpx_codec_destroy_fn_t)(

```

```

    vpx_codec_alg_priv_t *ctx);

/*!\brief parse stream info function pointer prototype
 *
 * Performs high level parsing of the bitstream. This function is
 * called by the generic vpx_codec_parse_stream() wrapper function,
 * so plugins implementing this interface may trust the input
 * parameters to be properly initialized.

```



```

*
* \param[in]      data      Pointer to a block of data to parse
* \param[in]      data_sz   Size of the data buffer
* \param[in,out]  si        Pointer to stream info to update. The size
*                             member \ref MUST be properly initialized,
*                             but \ref MAY be clobbered by the
*                             algorithm. This parameter \ref MAY be
*                             NULL.
*
* \retval #VPX_CODEC_OK
*         Bitstream is parsable and stream information updated
*/
typedef vpx_codec_err_t (*vpx_codec_peek_si_fn_t)(
    const uint8_t      *data,
    unsigned int        data_sz,
    vpx_codec_stream_info_t *si);

/*!\brief Return information about the current stream.
*
* Returns information about the stream that has been parsed during
* decoding.
*
* \param[in]      ctx        Pointer to this instance's context
* \param[in,out]  si        Pointer to stream info to update. The size
*                             member \ref MUST be properly initialized,
*                             but \ref MAY be clobbered by the
*                             algorithm. This parameter \ref MAY be
*                             NULL.
*
* \retval #VPX_CODEC_OK
*         Bitstream is parsable and stream information updated
*/
typedef vpx_codec_err_t (*vpx_codec_get_si_fn_t)(
    vpx_codec_alg_priv_t *ctx,
    vpx_codec_stream_info_t *si);

/*!\brief control function pointer prototype
*
* This function is used to exchange algorithm specific data with the
* decoder instance. This can be used to implement features specific

```

* to a particular algorithm.

```

*
* This function is called by the generic vpx_codec_control() wrapper
* function, so plugins implementing this interface may trust the
* input parameters to be properly initialized. However, this
* interface does not provide type safety for the exchanged data or
* assign meanings to the control codes. Those details should be
* specified in the algorithm's header file. In particular, the
* ctrl_id parameter is guaranteed to exist in the algorithm's
* control mapping table, and the data paramter may be NULL.
*
*
* \param[in]      ctx      Pointer to this instance's context
* \param[in]      ctrl_id  Algorithm specific control identifier
* \param[in,out]  data      Data to exchange with algorithm instance.
*
* \retval #VPX_CODEC_OK
*      The internal state data was deserialized.
*/
typedef vpx_codec_err_t (*vpx_codec_control_fn_t)(
    vpx_codec_alg_priv_t *ctx,
    int                  ctrl_id,
    va_list              ap);

/*!\brief control function pointer mapping
*
* This structure stores the mapping between control identifiers and
* implementing functions. Each algorithm provides a list of these
* mappings. This list is searched by the vpx_codec_control() wrapper
* function to determine which function to invoke. The special
* value {0, NULL} is used to indicate end-of-list, and must be
* present. The special value {0, <non-null>} can be used as a
* catch-all mapping. This implies that ctrl_id values chosen by the
* algorithm \ref MUST be non-zero.
*/
typedef const struct
{
    int                  ctrl_id;
    vpx_codec_control_fn_t  fn;
} vpx_codec_ctrl_fn_map_t;

/*!\brief decode data function pointer prototype
*
* Processes a buffer of coded data. If the processing results in a
* new decoded frame becoming available, #VPX_CODEC_CB_PUT_SLICE and
* #VPX_CODEC_CB_PUT_FRAME events are generated as appropriate. This
* function is called by the generic vpx_codec_decode() wrapper
* function, so plugins implementing this interface may trust the

```

```
* input parameters to be properly initialized.
*
* \param[in] ctx          Pointer to this instance's context
* \param[in] data          Pointer to this block of new coded data. If
*                          NULL, a #VPX_CODEC_CB_PUT_FRAME event is
*                          posted for the previously decoded frame.
* \param[in] data_sz       Size of the coded data, in bytes.
*
* \return Returns #VPX_CODEC_OK if the coded data was processed
*         completely and future pictures can be decoded without
*         error. Otherwise, see the descriptions of the other error
*         codes in ::vpx_codec_err_t for recoverability
*         capabilities.
*/
typedef vpx_codec_err_t (*vpx_codec_decode_fn_t)(
    vpx_codec_alg_priv_t *ctx,
    const uint8_t *data,
    unsigned int data_sz,
    void *user_priv,
    long deadline);

/*!\brief Decoded frames iterator
*
* Iterates over a list of the frames available for display. The
* iterator storage should be initialized to NULL to start the
* iteration. Iteration is complete when this function returns NULL.
*
* The list of available frames becomes valid upon completion of the
* vpx_codec_decode call, and remains valid until the next call to
* vpx_codec_decode.
*
* \param[in] ctx          Pointer to this instance's context
* \param[in out] iter      Iterator storage, initialized to NULL
*
* \return Returns a pointer to an image, if one is ready for
*         display. Frames produced will always be in PTS
*         (presentation time stamp) order.
*/
typedef vpx_image_t* (*vpx_codec_get_frame_fn_t)(
    vpx_codec_alg_priv_t *ctx,
    vpx_codec_iter_t *iter);

/*!\brief e_xternal Memory Allocation memory map get iterator
*
* Iterates over a list of the memory maps requested by the decoder.
```

- * The iterator storage should be initialized to NULL to start the
- * iteration. Iteration is complete when this function returns NULL.

```

*
* \param[in out] iter      Iterator storage, initialized to NULL
*
* \return Returns a pointer to an memory segment descriptor, or NULL
*         to indicate end-of-list.
*/
typedef vpx_codec_err_t (*vpx_codec_get_mmap_fn_t)(
    const vpx_codec_ctx_t      *ctx,
    vpx_codec_mmap_t           *mmap,
    vpx_codec_iter_t           *iter);

/*\brief e_xternal Memory Allocation memory map set iterator
*
* Sets a memory descriptor inside the decoder instance.
*
* \param[in] ctx          Pointer to this instance's context
* \param[in] mmap         Memory map to store.
*
* \retval #VPX_CODEC_OK
*         The memory map was accepted and stored.
* \retval #VPX_CODEC_MEM_ERROR
*         The memory map was rejected.
*/
typedef vpx_codec_err_t (*vpx_codec_set_mmap_fn_t)(
    vpx_codec_ctx_t           *ctx,
    const vpx_codec_mmap_t    *mmap);

typedef vpx_codec_err_t (*vpx_codec_encode_fn_t)(
    vpx_codec_alg_priv_t      *ctx,
    const vpx_image_t          *img,
    vpx_codec_pts_t           pts,
    unsigned long              duration,
    vpx_enc_frame_flags_t      flags,
    unsigned long              deadline);
typedef const vpx_codec_cx_pkt_t*(*vpx_codec_get_cx_data_fn_t)(
    vpx_codec_alg_priv_t      *ctx,
    vpx_codec_iter_t          *iter);

```

```

typedef vpx_codec_err_t
(*vpx_codec_enc_config_set_fn_t)(
    vpx_codec_alg_priv_t      *ctx,
    const vpx_codec_enc_cfg_t  *cfg);
typedef vpx_fixed_buf_t *
(*vpx_codec_get_global_headers_fn_t)(vpx_codec_alg_priv_t      *ctx);

typedef vpx_image_t *

```

```

(*vpx_codec_get_preview_frame_fn_t)(vpx_codec_alg_priv_t      *ctx);

/*!\brief usage configuration mapping
 *
 * This structure stores the mapping between usage identifiers and
 * configuration structures. Each algorithm provides a list of these
 * mappings. This list is searched by the
 * vpx_codec_enc_config_default() wrapper function to determine which
 * config to return. The special value {-1, {0}} is used to indicate
 * end-of-list, and must be present. At least one mapping must be
 * present, in addition to the end-of-list.
 *
 */
typedef const struct
{
    int                usage;
    vpx_codec_enc_cfg_t cfg;
} vpx_codec_enc_cfg_map_t;

#define NOT_IMPLEMENTED 0

/*!\brief Decoder algorithm interface interface
 *
 * All decoders \ref MUST expose a variable of this type.
 */
struct vpx_codec_iface
{
    const char          *name;
    int                 abi_version;
    vpx_codec_caps_t     caps;
    vpx_codec_init_fn_t  init;
    vpx_codec_destroy_fn_t destroy;

```

```

vpx_codec_ctrl_fn_map_t  *ctrl_maps;
vpx_codec_get_mmap_fn_t  get_mmap;
vpx_codec_set_mmap_fn_t  set_mmap;
struct
{
    vpx_codec_peek_si_fn_t  peek_si;
    vpx_codec_get_si_fn_t   get_si;
    vpx_codec_decode_fn_t   decode;
    vpx_codec_get_frame_fn_t get_frame;
} dec;
struct
{
    vpx_codec_enc_cfg_map_t      *cfg_maps;
    vpx_codec_encode_fn_t        encode;
    vpx_codec_get_cx_data_fn_t   get_cx_data;
    vpx_codec_enc_config_set_fn_t cfg_set;

```

```

    vpx_codec_get_global_headers_fn_t  get_glob_hdrs;
    vpx_codec_get_preview_frame_fn_t   get_preview;
} enc;
};

/*!\brief Callback function pointer / user data pair storage */
typedef struct vpx_codec_priv_cb_pair
{
    union
    {
        vpx_codec_put_frame_cb_fn_t  put_frame;
        vpx_codec_put_slice_cb_fn_t  put_slice;
    };
    void *user_priv;
} vpx_codec_priv_cb_pair_t;

/*!\brief Instance private storage
*
* This structure is allocated by the algorithm's init function. It
* can be extended in one of two ways. First, a second, algorithm
* specific structure can be allocated and the priv member pointed to
* it. Alternatively, this structure can be made the first member of
* the algorithm specific structure, and the pointer casted to the
* proper type.

```

```

*/
struct vpx_codec_priv
{
    unsigned int                sz;
    vpx_codec_iface_t          *iface;
    struct vpx_codec_alg_priv  *alg_priv;
    const char                  *err_detail;
    vpx_codec_flags_t          init_flags;
    struct
    {
        vpx_codec_priv_cb_pair_t  put_frame_cb;
        vpx_codec_priv_cb_pair_t  put_slice_cb;
    } dec;
    struct
    {
        int                        tbd;
        struct vpx_fixed_buf       cx_data_dst_buf;
        unsigned int               cx_data_pad_before;
        unsigned int               cx_data_pad_after;
        vpx_codec_cx_pkt_t         cx_data_pkt;
    } enc;
};

```

```

#undef VPX_CTRL_USE_TYPE
#define VPX_CTRL_USE_TYPE(id, typ) \
    static typ id##_value(va_list args) \
    {return va_arg(args, typ);} \
    static typ id##_convert(void *x)\
    {\
        union\
        {\
            void *x;\
            typ  d;\
        } u;\
        u.x = x;\
        return u.d;\
    }

```

```

#undef VPX_CTRL_USE_TYPE_DEPRECATED
#define VPX_CTRL_USE_TYPE_DEPRECATED(id, typ) \

```

```

static typ id##_value(va_list args) \
{return va_arg(args, typ);} \
static typ id##_convert(void *x)\
{\
    union\
    {\
        void *x;\
        typ    d;\
    } u;\
    u.x = x;\
    return u.d;\
}

#define CAST(id, arg) id##_value(arg)
#define RECAST(id, x) id##_convert(x)

/* Internal Utility Functions
 *
 * The following functions are indented to be used inside algorithms
 * as utilities for manipulating vpx_codec_* data structures.
 */
struct vpx_codec_pkt_list
{
    unsigned int      cnt;
    unsigned int      max;
    struct vpx_codec_cx_pkt pkts[1];
};

#define vpx_codec_pkt_list_decl(n)\

```

```

    union {struct vpx_codec_pkt_list head;\
        struct {struct vpx_codec_pkt_list head;\
            struct vpx_codec_cx_pkt    pkts[n];} alloc;}

#define vpx_codec_pkt_list_init(m)\
    (m)->alloc.head.cnt = 0,\
    (m)->alloc.head.max = \
    sizeof((m)->alloc.pkts) / sizeof((m)->alloc.pkts[0])

int
vpx_codec_pkt_list_add(struct vpx_codec_pkt_list *,

```



```

        const struct vpx_codec_cx_pkt *);

const vpx_codec_cx_pkt_t*
vpx_codec_pkt_list_get(struct vpx_codec_pkt_list *list,
                      vpx_codec_iter_t *iter);

#include <stdio.h>
#include <setjmp.h>
struct vpx_internal_error_info
{
    vpx_codec_err_t  error_code;
    int              has_detail;
    char             detail[80];
    int              setjmp;
    jmp_buf          jmp;
};

static void vpx_internal_error(struct vpx_internal_error_info *info,
                              vpx_codec_err_t error,
                              const char *fmt,
                              ...)
{
    va_list ap;

    info->error_code = error;
    info->has_detail = 0;

    if (fmt)
    {
        size_t sz = sizeof(info->detail);

        info->has_detail = 1;
        va_start(ap, fmt);
        vsnprintf(info->detail, sz - 1, fmt, ap);
        va_end(ap);
        info->detail[sz-1] = '\\0';
    }

    if (info->setjmp)
        longjmp(info->jmp, info->error_code);
}

```

```
}  
#endif
```

----- End code block -----

[20.20.](#) vpx_decoder.h

----- Begin code block -----

```
/*  
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.  
 *  
 * Use of this source code is governed by a BSD-style license  
 * that can be found in the LICENSE file in the root of the source  
 * tree. An additional intellectual property rights grant can be  
 * found in the file PATENTS. All contributing project authors may  
 * be found in the AUTHORS file in the root of the source tree.  
 */  
  
/*!\defgroup decoder Decoder Algorithm Interface  
 * \ingroup codec  
 * This abstraction allows applications using this decoder to easily  
 * support multiple video formats with minimal code duplication. This  
 * section describes the interface common to all decoders.  
 * @{  
 */  
  
/*!\file vpx_decoder.h  
 * \brief Describes the decoder algorithm interface to applications.  
 *  
 * This file describes the interface between an application and a  
 * video decoder algorithm.  
 *  
 */  
#ifdef __cplusplus  
extern "C" {  
#endif  
  
#ifndef VPX_DECODER_H  
#define VPX_DECODER_H  
#include "vpx_codec.h"
```

```
/*!\brief Current ABI version number
*
* \internal
* If this file is altered in any way that changes the ABI, this
* value must be bumped. Examples include, but are not limited
* to, changing types, removing or reassigning enums,
* adding/removing/rearranging fields to structures
*/
#define VPX_DECODER_ABI_VERSION (2 + VPX_CODEC_ABI_VERSION)

/*!\brief Decoder capabilities bitfield
*
* Each decoder advertises the capabilities it supports as part
* of its ::vpx_codec_iface_t interface structure. Capabilities
* are extra interfaces or functionality, and are not required
* to be supported by a decoder.
*
* The available flags are specified by VPX_CODEC_CAP_* defines.
*/
#define VPX_CODEC_CAP_PUT_SLICE 0x10000 /**< Will issue put_slice
callbacks */
#define VPX_CODEC_CAP_PUT_FRAME 0x20000 /**< Will issue put_frame
callbacks */
#define VPX_CODEC_CAP_POSTPROC 0x40000 /**< Can postprocess decoded
frame */

/*!\brief Initialization-time Feature Enabling
*
* Certain codec features must be known at initialization time,
* to allow for proper memory allocation.
*
* The available flags are specified by VPX_CODEC_USE_* defines.
*/
#define VPX_CODEC_USE_POSTPROC 0x10000 /**< Postprocess decoded
frame */

/*!\brief Stream properties
*
* This structure is used to query or set properties of the
* decoded stream. Algorithms may extend this structure with data
* specific to their bitstream by setting the sz member
* appropriately.
*/
typedef struct vpx_codec_stream_info
{
    unsigned int sz;    /**< Size of this structure */
    unsigned int w;     /**< Width (or 0 for unknown/default) */
}
```

```
    unsigned int h;    /**< Height (or 0 for unknown/default) */
```

```
    unsigned int is_kf; /**< Current frame is a keyframe */
} vpx_codec_stream_info_t;

/* REQUIRED FUNCTIONS
 *
 * The following functions are required to be implemented for all
 * decoders. They represent the base case functionality expected
 * of all decoders.
 */

/*!\brief Initialization Configurations
 *
 * This structure is used to pass init time configuration options
 * to the decoder.
 */
typedef struct vpx_codec_dec_cfg
{
    unsigned int threads; /**< Maximum number of threads to use,
        default 1 */
    unsigned int w;    /**< Width */
    unsigned int h;    /**< Height */
} vpx_codec_dec_cfg_t; /**< alias for struct vpx_codec_dec_cfg */

/*!\brief Initialize a decoder instance
 *
 * Initializes a decoder context using the given interface.
 * Applications should call the vpx_codec_dec_init convenience
 * macro instead of this function directly, to ensure that the
 * ABI version number parameter is properly initialized.
 *
 * In XMA mode (activated by setting VPX_CODEC_USE_XMA in the
 * flags parameter), the storage pointed to by the cfg parameter
 * must be kept readable and stable until all memory maps have
 * been set.
 *
 * \param[in]    ctx    Pointer to this instance's context.
 * \param[in]    iface  Pointer to the algorithm interface to
 *                      use.
```

```

* \param[in]    cfg      Configuration to use, if known. May be
*                  NULL.
* \param[in]    flags    Bitfield of VPX_CODEC_USE_* flags
* \param[in]    ver      ABI version number. Must be set to
*                  VPX_DECODER_ABI_VERSION
* \retval #VPX_CODEC_OK
*          The decoder algorithm initialized.
* \retval #VPX_CODEC_MEM_ERROR

```

```

*      Memory allocation failed.
*/
vpx_codec_err_t vpx_codec_dec_init_ver(
    vpx_codec_ctx_t      *ctx,
    vpx_codec_iface_t     *iface,
    vpx_codec_dec_cfg_t   *cfg,
    vpx_codec_flags_t     flags,
    int                   ver);

/*!\brief Convenience macro for vpx_codec_dec_init_ver()
*
* Ensures the ABI version parameter is properly set.
*/
#define vpx_codec_dec_init(ctx, iface, cfg, flags) \
    vpx_codec_dec_init_ver(ctx, iface, cfg, flags, \
        VPX_DECODER_ABI_VERSION)

/*!\brief Parse stream info from a buffer
*
* Performs high level parsing of the bitstream. Construction of
* a decoder context is not necessary. Can be used to determine
* if the bitstream is of the proper format, and to extract
* information from the stream.
*
* \param[in]    iface    Pointer to the algorithm interface
* \param[in]    data      Pointer to a block of data to parse
* \param[in]    data_sz   Size of the data buffer
* \param[in,out] si       Pointer to stream info to update. The
*                          size member
*                          \ref MUST be properly initialized, but
*                          \ref MAY be clobbered by the
*                          algorithm. This parameter \ref MAY be

```

```

*                                     NULL.
*
* \retval #VPX_CODEC_OK
*     Bitstream is parsable and stream information updated
*/
vpx_codec_err_t vpx_codec_peek_stream_info(
    vpx_codec_iface_t    *iface,
    const uint8_t        *data,
    unsigned int          data_sz,
    vpx_codec_stream_info_t *si);

/*!\brief Return information about the current stream.
*
* Returns information about the stream that has been parsed

```

```

* during decoding.
*
* \param[in]    ctx    Pointer to this instance's context
* \param[in,out] si    Pointer to stream info to update. The
*                       size member \ref MUST be properly
*                       initialized, but \ref MAY be clobbered
*                       by the algorithm. This parameter \ref
*                       MAY be NULL.
*
* \retval #VPX_CODEC_OK
*     Bitstream is parsable and stream information updated
*/
vpx_codec_err_t vpx_codec_get_stream_info(
    vpx_codec_ctx_t    *ctx,
    vpx_codec_stream_info_t *si);

/*!\brief Decode data
*
* Processes a buffer of coded data. If the processing results in
* a new decoded frame becoming available, PUT_SLICE and
* PUT_FRAME events may be generated, as appropriate. Encoded
* data \ref MUST be passed in DTS (decode time stamp) order.
* Frames produced will always be in PTS (presentation time
* stamp) order.
*

```

```

* \param[in] ctx          Pointer to this instance's context
* \param[in] data          Pointer to this block of new coded
*                          data. If NULL, a
*                          VPX_CODEC_CB_PUT_FRAME event is posted
*                          for the previously decoded frame.
* \param[in] data_sz       Size of the coded data, in bytes.
* \param[in] user_priv     Application specific data to associate
*                          with this frame.
* \param[in] deadline     Soft deadline the decoder should
*                          attempt to meet, in us. Set to zero
*                          for unlimited.
*
* \return Returns #VPX_CODEC_OK if the coded data was processed
*         completely and future pictures can be decoded without
*         error. Otherwise, see the descriptions of the other
*         error codes in ::vpx_codec_err_t for recoverability
*         capabilities.
*/
vpx_codec_err_t vpx_codec_decode(vpx_codec_ctx_t      *ctx,
                                const uint8_t         *data,
                                unsigned int           data_sz,
                                void                   *user_priv,

```

```

                                long                deadline);

```

```

/*!\brief Decoded frames iterator

```

```

*
* Iterates over a list of the frames available for display. The
* iterator storage should be initialized to NULL to start the
* iteration. Iteration is complete when this function returns
* NULL.
*
* The list of available frames becomes valid upon completion of
* the vpx_codec_decode call, and remains valid until the next
* call to vpx_codec_decode.
*
* \param[in]      ctx      Pointer to this instance's context
* \param[in,out] iter      Iterator storage, initialized to NULL
*
* \return Returns a pointer to an image, if one is ready for
*         display. Frames produced will always be in PTS

```

```

*          (presentation time stamp) order.
*/
vpx_image_t *vpx_codec_get_frame(vpx_codec_ctx_t *ctx,
                                vpx_codec_iter_t *iter);

/*!\defgroup cap_put_frame Frame-Based Decoding Functions
*
* The following functions are required to be implemented for all
* decoders that advertise the VPX_CODEC_CAP_PUT_FRAME
* capability. Calling these functions for codecs that don't
* advertise this capability will result in an error code being
* returned, usually VPX_CODEC_ERROR
* @{
*/

/*!\brief put frame callback prototype
*
* This callback is invoked by the decoder to notify the
* application of the availability of decoded image data.
*/
typedef void (*vpx_codec_put_frame_cb_fn_t)(
    void *user_priv,
    const vpx_image_t *img);

/*!\brief Register for notification of frame completion.
*
* Registers a given function to be called when a decoded frame

```

```

* is available.
*
* \param[in] ctx      Pointer to this instance's context
* \param[in] cb       Pointer to the callback function
* \param[in] user_priv User's private data
*
* \retval #VPX_CODEC_OK
*      Callback successfully registered.
* \retval #VPX_CODEC_ERROR
*      Decoder context not initialized, or algorithm not capable
*      of posting slice completion.
*/

```



```

vpx_codec_err_t vpx_codec_register_put_frame_cb(
    vpx_codec_ctx_t          *ctx,
    vpx_codec_put_frame_cb_fn_t cb,
    void                    *user_priv);

/*!@} - end defgroup cap_put_frame */

/*!\defgroup cap_put_slice Slice-Based Decoding Functions
 *
 * The following functions are required to be implemented for all
 * decoders that advertise the VPX_CODEC_CAP_PUT_SLICE
 * capability. Calling these functions for codecs that don't
 * advertise this capability will result in an error code being
 * returned, usually VPX_CODEC_ERROR
 * @{
 */

/*!\brief put slice callback prototype
 *
 * This callback is invoked by the decoder to notify the
 * application of the availability of partially decoded image
 * data.
 */
typedef void (*vpx_codec_put_slice_cb_fn_t)(
    void          *user_priv,
    const vpx_image_t *img,
    const vpx_image_rect_t *valid,
    const vpx_image_rect_t *update);

/*!\brief Register for notification of slice completion.
 *
 * Registers a given function to be called when a decoded slice
 * is available.
 */

```

```

* \param[in] ctx      Pointer to this instance's context
* \param[in] cb       Pointer to the callback function
* \param[in] user_priv User's private data
*
* \retval #VPX_CODEC_OK

```

```

    *      Callback successfully registered.
    * \retval #VPX_CODEC_ERROR
    *      Decoder context not initialized, or algorithm not capable
    *      of posting slice completion.
    */
vpx_codec_err_t vpx_codec_register_put_slice_cb(
    vpx_codec_ctx_t          *ctx,
    vpx_codec_put_slice_cb_fn_t cb,
    void                    *user_priv);

/*!@} - end defgroup cap_put_slice*/

/*!@} - end defgroup decoder*/

#endif

#ifdef __cplusplus
}
#endif

#if !defined(VPX_CODEC_DISABLE_COMPAT) || !VPX_CODEC_DISABLE_COMPAT
#include "vpx_decoder_compat.h"
#endif

```

---- End code block -----

[20.21.](#) vpx_integer.h

---- Begin code block -----

```

/*
 * Copyright (c) 2010 The VP8 project authors. All Rights Reserved.
 *
 * Use of this source code is governed by a BSD-style license
 * that can be found in the LICENSE file in the root of the source
 * tree. An additional intellectual property rights grant can be
 * found in the file PATENTS. All contributing project authors may
 * be found in the AUTHORS file in the root of the source tree.
 */

```

```

#ifndef VPX_INTEGER_H
#define VPX_INTEGER_H

/* get ptrdiff_t, size_t, wchar_t, NULL */
#include <stddef.h>

#if defined(_MSC_VER) || defined(VPX_EMULATE_INTTYPES)
typedef signed char   int8_t;
typedef signed short  int16_t;
typedef signed int    int32_t;

typedef unsigned char  uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int   uint32_t;

#if defined(_MSC_VER)
typedef signed __int64   int64_t;
typedef unsigned __int64 uint64_t;
#define PRId64 "I64d"
#endif

#ifdef HAVE_ARMV6
typedef unsigned int int_fast16_t;
#else
typedef signed short int_fast16_t;
#endif
typedef signed char  int_fast8_t;
typedef unsigned char uint_fast8_t;

#ifndef _UINTPTR_T_DEFINED
typedef unsigned int  uintptr_t;
#endif

#else

/* Most platforms have the C99 standard integer types. */

#if defined(__cplusplus) && !defined(__STDC_FORMAT_MACROS)
#define __STDC_FORMAT_MACROS
#endif
#include <stdint.h>
#include <inttypes.h>

#endif

#endif

```

---- End code block -----

[21](#). References

[ITU-R_BT.601]

International Telecommunication Union, "ITU BT.601: Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios", January 2007.

[Bell]

Bell, T., Cleary, J., and I. Witten, "Text Compression", 1990.

[Kernighan]

Kernighan, B. and D. Ritchie, "The C Programming Language (2nd edition)", April 1988.

[Loeffler]

Loeffler, C., Ligtenberg, A., and G. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications", May 1989.

[Shannon]

Shannon, C., "A Mathematical Theory of Communication", Bell System Technical Journal Vol. 27, pp. 379-423, 623-656, July, October 1948.

Authors' Addresses

James Bankoski
Google, Inc.

Email: jimbankoski@google.com

Paul Wilkins
Google, Inc.

Email: paulwilkins@google.com

Yaowu Xu
Google, Inc.

Email: yaowu@google.com

