ON2 TECHNOLOGIES, INC.

OVERVIEW

VP7 Data Format and Decoder

March 28, 2005

Document version: 1.5

On2 Technologies, Inc.
21 Corporate Dr.
Suite 103
Clifton Park, NY 12065
www.on2.com

Contents

1. INTRODUCTION

This document describes the VP7 compressed video data created by On2 Technologies Inc. together with a discussion of the decoding procedure for this format. It is intended to be used in conjunction with, and as a guide to, the reference decoder provided by On2 Technologies.

Like many contemporary video compression schemes, VP7 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 the discrete cosine transform (hereafter abbreviated as 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 facilitates 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.

VP7 augments these basic concepts with, among many things, sophisticated usage of contextual probabilities. The result is a significant reduction in datarate at a given quality when compared to any other extant video compression algorithm.

Unlike some similar schemes (the older MPEG formats, for example), VP7 specifies exact values for reconstructed pixels. Specifically, the specification for the DCT portion 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 with 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. UNCOMPRESSED FRAME FORMAT

VP7 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; the 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.

Occasionally, at very low datarates, a compression system may decide to reduce the resolution of the input signal to facilitate compression efficiency. The VP7 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 of course restores the video frames to their original resolution. Put another way, viewing the compression/decompression system as a "black box," while the input and output to this system is always at the given resolution, the compressor may decide to "cheat" and process the signal at a lower resolution; the decompressor then needs the ability to restore the signal to its original resolution.

Internally, VP7 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, 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 will be 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 VP7 algorithm. In particular, the 2x2 chroma blocks corresponding to a 4x4 Y subblock are never treated explicitly either in the data format or in the algorithm specification.

The DCT always operates at 4x4 resolution, both on subblocks and (sometimes) on the 4x4 array of average intensities of the 16 Y subblocks of a macroblock. These average intensities are, up to a constant normalization factor, nothing more that the zeroth DCT coefficients of the Y subblocks and this "higher-level" DCT is a substitute for the explicit specification of these coefficients, in exactly the same way as the DCT of a subblock substitutes for the specification of the pixel values comprising the subblock. We sometimes 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.

Pushing the chemical analogy perhaps too far, pixels can then be thought of as "particles" and macroblocks as "molecules."

The frame layout used by the reference decoder can be found in the file "yv12config.h."

3. COMPRESSED FRAME TYPES

VP7 uses a different model of interframe prediction than some other compression strategies.

In the first place, there are only two types of frames in VP7.

Intra frames (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.

Inter frames (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, then, 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 backward or bidirectional prediction. No frame is predicted using frames temporally subsequent to it; there is no analogue to an MPEG B-frame.

Secondly, VP7 augments these notions with that of alternate prediction frames, called “golden frames.” Blocks in an interframe may be predicted using blocks in the immediately previous frame as well as the most recent golden frame. Every key frame is automatically golden and any interframe may optionally replace the most recent golden frame. As will be detailed later, partial modifications of golden frames (with arbitrary 16x16 subblocks of an interframe) are also supported. Golden frames provide much of the benefit of bidirectional prediction (such as reuse of well-encoded frames or sections of frames) while avoiding the conceptual and logistic difficulty of out-of-order frame transmission.

Golden 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 VP7 decompressor 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 provided by the playback environment and are irrelevant to the decoding process itself.

The decoder is simply presented with a sequence of compressed frames and produces, for each compressed frame, a decompressed (reconstructed) YVU frame corresponding to the compressed frame. As was stated in the introduction, the exact pixel values in the reconstructed frame are part of VP7’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 depending on all of its predecessors up to and including 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 decodings of prior such sequences that may already have occurred.

At the highest level, every compressed frame has three pieces. It begins with a 4-byte frame tag and is followed by two blocks of compressed data (called “partitions”). These compressed data partitions begin and end on byte boundaries.

The frame tag contains three fields:

1. A 1-bit frame type (0 for key frames, 1 for interframes).
2. A 3-bit version number (0 for now, may increase with future incompatible variants of the VP7 data format).
3. A 20-bit field containing the size of the first data partition in bytes.

Note that there are 8 unused bits in the frame tag. This allows the tag and partitions to have byte lengths (and addresses in some systems) that are evenly divisible by 4, which can improve efficiency.

The first 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 was stated above, the macroblock-level information occurs in raster-scan order.

The second partition contains, for each block, again in scan order, the DCT 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 data rate.

This partitioning of the data 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.

Both partitions are decoded using (separate instances of) the boolean entropy decoder described below. 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 4-byte frame tag described above is there such a physical correspondence.

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

While this document will of course specify the ordering of the partition data correctly, the details and semantics of this data will be discussed in a more "logical" fashion that is hoped will facilitate comprehension. For example, the frame header contains updates to many probability tables used in decoding per-macroblock data. The latter will often be 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 VP7 decoder needs to maintain three 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, and the most recent golden frame.

Most implementations will wish to "pad" these buffers with "invisible" pixels that extend a moderate number of pixels in the Y (U or V) plane beyond all four edges of the visible image. This simplifies interframe prediction by allowing (virtually) all 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 in the discussion of motion vectors, interframe prediction, and subpixel interpolation below..

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, we 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 by 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 parts. The first ("prediction" or "mode") part comes in the remainder of the first data partition. The second ("residue" or "DCT") part comprises the second data partition. For each macroblock, the prediction data must be processed before the residue.

Each macroblock is predicted using one (and only one) of three 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 either the previous frame or the golden 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 16 Y subblocks are expressed via a 25th DCT 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. (and 2) Copying the row (or column) throughout the prediction buffer.
2. Copying the average value of the row and column .
3. Extrapolation from the row and column using the (fixed) second difference (horizontal and vertical) from the upper left corner.

Additionally, the 16 Y subblocks may be predicted independently of each other using one of ten different "modes," four of which are essentially the same as those described above, together with six diagonal prediction methods. This is the only type of prediction (inter modes included) for which the residue signal does not use the Y2 block to encode the DC portion of the sixteen 4x4 Y subblock DCTs. Also, 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 or the golden 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, in the end, each of the 16 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 VP7 have quarter-pixel precision (1/8 pixel in the chroma planes). 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 or golden frame are simply copied into the correct position of the prediction buffer.

Fractional displacements are conceptually, and implementation-wise, 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 VP7 provides a good balance of robustness, accuracy, and efficiency.

Leaving the details for the implementation discussion, the pixel interpolation is calculated by taking a weighted average (using reasonable-precision integer math) of 3 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 8 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 4 times the area) of luma pixels, the calculated chroma motion vectors have 1/8 pixel resolution but the procedure of 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 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 reason for the terminology “loop filter” is that this filter is part of the “coding loop,” that is, it effects the last-frame and golden frame buffers that are used to predict ensuing frames. This is distinguished from the “post-processing” filters discussed earlier which affect only the viewed video and do not “feed into” subsequent frames.

Next, if signaled in the data, the current frame (or individual macroblocks within the current frame) may replace all (or some of) the golden frame prediction buffer.

The “halos” of the last frame and golden 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 regard to 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, we will have little to say about these issues.

While the basic ideas of prediction and correction used by VP7 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 modifications 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 more-or-less concludes our top-level discussion of decoding and we next discuss the individual aspects in more depth.

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

6. DESCRIPTION OF ALGORITHMS

As the intent of the reference decoder source code, supplemented by this document, is to specify a platform-independent procedure for the decoding and reconstruction of a VP7 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 VP7 decoder is written in “C,” these algorithm fragments are written using the “C” programming language, augmented with a few simple definitions below.

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

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

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

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

All VP7 decoding algorithms use integer math. To facilitate specification of arithmetic precision, we define the following types.

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

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

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

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

typedef unsigned int32 uint32;
```

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

typedef unsigned int uint;


/*  Pixel arithmetic often occurs at 16- or 32-bit wide precision and the
results need to be "saturated" or clamped to an 8-bit range. */

uint8 clamp255( int v) { return v < 0? 0 : (v < 255? v : 255);}


/*  As is elaborated in the discussion of the BoolDecoder below, VP7
represents probabilities as unsigned 8-bit numbers. */

typedef uint8 Prob;
```

We occasionally need to discuss mathematical functions involving honest-to-goodness "infinite precision" real numbers. The DCT is first described via the cosine function "cos," the ratio of the lengths of the circumference and diameter of a circle is denoted "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 nonnegative integer, pow(2, y) may be expressed in C as "1 << y."

7. BOOLEAN ENTROPY DECODER

As discussed in the overview above, essentially the entire VP7 data stream is encoded using a boolean entropy coder.

An understanding of the BoolDecoder is critical to the implementation of a VP7 decompressor, so we discuss it at some length. It is easier to comprehend the BoolDecoder in conjunction with the BoolEncoder used by the compressor to write the compressed data partitions.

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

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

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

If we encode a sequence of bools whose probability of being zero is p (and whose probability of being 1 is 1-p), the lowest possible datarate per value is p*log(p) + (1-p)*log(1-p); taking the logarithms to the base ½ expresses the datarate in bits/value.

We give two simple examples. At one extreme, if p=½, then log(p) = log(1-p) = 1 and the lowest possible datarate per bool is ½ + ½ = 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 VP7 datastream have zero-probabilities nowhere near ½, the compression provided by the BoolEncoder is critical to the performance of VP7.

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 <= x < 1. The bits (or bytes) in x are of course written from high to low order and if b[j] (B[j]) is the j^{th} bit (byte) in the partition, the value x is simply the sum (starting with j = 1) of pow(2, -j) * b[j] or 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 p1 is the probability of the first bool being zero and a zero is coded, the range of possible x is restricted to 0 <= x < p1. If a one is coded, the range becomes p1 <= x < 1.

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

Assuming only finitely many values are to be coded, after the encoder has received the last bool, it can simply write as its output any value x that lies in the final interval. VP7 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.

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 coded probabilities were correct, 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

VP7’s bool coder works with 8-bit probabilities p. The range of such p is 0 <= p <= 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 32-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 bit stream 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 output is the same thing as adding pow(2,-n) to x.

Together with the bit position, the encoder must maintain two unsigned 8-bit numbers which we call "bottom" and "range." Writing "w" for the n bits already written and S = pow(2, -n-9) 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 * bottom)  <=  v  < w + (S * (bottom + 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 <= range <= 255.

The process for encoding a boolean value "val" whose probability of being zero is prob/256 (and whose probability of being one is (256-prob)/256), with 1 <= prob <= 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 + ( (prob * (range-1)) >> 8);
```

"split" is approximately (prob/256) * range and lies within the bounds 1 <= split <= range-1. These bounds ensure the correctness of the decoding procedure described below.

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

Regardless of the value encoded, "range" has been reduced and now has the bounds 1 <= range <= 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 <= range <= 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 bit stream.

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 8 input bits into "value." The decoder maintains "range" and calculates "split" in exactly the same way as does the encoder.

To decode a bool, it compares "value" to "split"; if value < split, the bool is zero, and range is replaced with split. If value >= split, the bool is one, range is replaced with range – split, and value is replaced with value – split.

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

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

7.3 Actual Implementation

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

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

```
/* Encoder first */

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


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

void initBoolEncoder( BoolEncoder *e, uint8 *startPartition)
{
   e->output = startPartition;
   e->range = 255;
   e->bottom = 0;
   e->bitCount = 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 addOneToOutput( uint8 *q)
{
   while( *--q == 255)
      *q = 0;
   ++*q;
}

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

void writeBool( BoolEncoder *e, uint8 prob, int boolValue)
{
   /* 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( boolValue) {
      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 */
         addOneToOutput( e->output);

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

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

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

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

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

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

void flushBoolEncoder( BoolEncoder *e)
{
   int c = e->bitCount;
   uint32 v = e->bottom;

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


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

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


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

```
void initBoolDecoder( BoolDecoder *d, uint8 *startPartition)
{
   {
      int i = 0;
      d->value = 0;        /* value = first 4 input bytes */
      while( ++i <= 4)
         d->value = (d->value << 8)  |  *startPartition++;
   }
   d->input = startPartition; /* ptr to next byte to be read */
   d->range = 255;            /* initial range is full */
   d->bitCount = 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 readBool( BoolDecoder *d, uint8 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 << 24;

   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->bitCount == 8) {  /* shift in new bits 8 at a time */
         d->bitCount = 0;
         d->value |= *d->input++;
      }
   }
   return retval;
}


/* Convenience function reads a "literal," that is, a "numBits" wide
   unsigned value whose bits come high- to low-order, with each bit
   encoded at probability 128 (i.e., 1/2). */

uint32 readLiteral( BoolDecoder *d, int numBits)
{
   uint32 v = 0;
   while( numBits--)
      v = (v << 1) + readBool( d, 128);
```

```
   return v;
}


/* Variant reads a signed number */

int32 readSignedLiteral( BoolDecoder *d, int numBits)
{
   int32 v = 0;
   if( !numBits)
      return 0;
   if( readBool( d, 128))
      v = -1;
   while( --numBits)
      v = (v << 1) + readBool( d, 128);
   return v;
}
```

8. BASIC DATA COMPONENTS

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 decoder "d," reading (as always) from its current reference point.

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

T	Tree-encoded value from small alphabet.

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

This is done by representing the alphabet as the leaves of a small binary tree. The (non-leaf) nodes of the tree have associated probabilities “p” and correspond to calls to readBool(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 readBool.

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 VP7 are chosen to (on average) minimize the number of calls to readBool. 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 readBool, and will similarly realize that the coding method described here never results in a higher datarate than the Huffman method and, indeed, often results in a much lower datarate. 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 VP7.

It is most convenient to represent the values using small positive integers, typically an “enum” counting up from zero. The largest alphabet that is tree-coded by VP7 has only 12 values (this happens to be the alphabet used to encode DCT coefficients). 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.

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

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

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

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

typedef int8 TreeIndex;
typedef const TreeIndex Tree[];

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

```
int TreedRead(
   BoolDecoder * const d,     /* BoolDecoder always returns a 0 or 1 */
   Tree t,                 /* tree specification */
   const Prob p[]    /* corresponding interior node probabilities */
) {
   register TreeIndex i = 0;  /* begin at root */

   /* Descend tree until leaf is reached */

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

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

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 VP7 of the same dataset being coded by different trees under different circumstances. The UV (or chroma) modes are a proper subset of the Y modes and, as such, have their own decoding tree.

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

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

   numUVmodes = B_PRED,  /* first four modes apply to chroma */
   numYmodes   /* all modes apply to luma */
}
intraMBmode;


/* 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 TreeIndex YmodeTree [2 * (numYmodes - 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 TreeIndex kfYmodeTree [2 * (numYmodes - 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 TreeIndex UVmodeTree [2 * (numUVmodes - 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 BoolDecoder d, a Y mode might be decoded as follows.*/

const Prob pretendItsHuffman [numYmodes - 1] = { 128, 128, 128, 128};

   Ymode = (intraMBmode) TreedRead( d, YmodeTree, pretendItsHuffman);
```

Tree-based decoding is implemented in the reference decoder file "treeReader.h."

9. FRAME HEADER

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."

A. Dimension information (keyframes only)

L(12)	width of Y plane in pixels
L(12)	height of Y plane in pixels
L(2)	2-bit horizontal scaling specification
L(2)	2-bit vertical scaling specification

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

The scaling specifications for each dimension are encoded as follows.

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 the decoding overview, allocation (or re-allocation) of data structures (such as the reconstruction buffer) whose size depends on dimension will be triggered here.

The dimension information does not appear in interframes.

B. Decoding information for all four macroblock-level features

Contains probability and value information for various macroblock-level overrides 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. The layout and semantics of this section will be described as part of the discussion of macroblock-level decoding and prediction.

C. Dequantization indices

All residue signals are specified via a quantized 4x4 DCT applied to the Y, U, V, or Y2 subblocks of a macroblock. Before inverting the transform, each decoded coefficient is multiplied by one of 6 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 6 values are specified using 7-bit indices into 6 corresponding fixed tables; the tables can be found in the reference decoder file "quant_common.c."

The first 7-bit index gives the dequantization table index for Y plane AC coefficients, called "YacQi." It is always coded and acts as a default for the other 5 quantization indices, which are named similarly. Each of these other indices may either default to YacQi or be specified literally according to a flag. Pseudocode to read the indices is then as follows.

```
YacQi = L(7);               /* Y ac index always specified */
YdcQi = F? L(7) : YacQi;    /* Y dc index specified if next flag is true */

Y2dcQi = F? L(7) : YacQi;   /* Y2 dc "" */
Y2acQi = F? L(7) : YacQi;   /* Y2 ac "" */

UVdcQi = F? L(7) : YacQi;   /* chroma dc "" */
UVacQi = F? L(7) : YacQi;   /* chroma ac "" */
```

By the way, our "pseudocode" is not so far from actual code. The expression "YdcQi = F? L(7) : YacQi;" is just an abbreviation for

```
YdcQi = readBool( d, 128) ? readLiteral( d, 7) : YacQi;
```

and so on.

D. Golden frame update flag (a Flag) for interframes only

If true, the golden frame is replaced by the current frame after reconstruction is complete. This flag does not occur for key frames, which always update the golden frame.

E. Fading information for previous frame

The immediately previous frame may be modified (before being used for prediction) via a simple "fading" transform, specified as follows.

F	Read next two fields and apply fading if 1
SignedLit(8)	"alpha" = value to add to prediction pixels
SignedLit(8)	"beta" = value by which to fade prediction pixels.

The fading algorithm is very simple and applies to the Y plane only. Each 8-bit Y pixel "y" is replaced with

```
clamp255( y + ((y * beta) >> 8) + alpha);
```

Note that fading never affects the golden frame. Note also that the Flag does occur (and is necessarily zero) in key frames, which of course do not reference the previous frame.

F. Loop filter type

A single Flag. Zero means “normal” loop filtering; one means “simple” loop filtering. Loop filtering occurs very late in the decoding process and is detailed below.

G. DCT coefficient ordering specification

This field begins with a Flag. If zero, we retain the coefficient order used on the previous frame.

If true, there follows 15 four-bit literals specifying the order in which the AC dct coefficients occur in the residue signal for the Y and chroma planes. Every possible index (from 1 to 15) occurs exactly once.

Regardless of the ordering of the AC coefficients, every DCT begins with the zeroth (DC) coefficient.

The second-order Y2 plane always uses the default scan order and, for the Y and chroma planes, the current coefficient order must be set to the default on every key frame (before decoding this field).

Here is the default scan order:

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

Each index in the table above (and in the bitstream when replacing the scan order) is of the form (row << 2) + column, where “row” is the vertical frequency index and “column” is the horizontal frequency index.

The scan order gives the two-dimensional frequency index as a function of coding position. Inverting this, that is, giving the coding position as a function of frequency index, shows that the scan order actually “zig-zags” through the coefficients:

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

The reason for the default “zig-zag” order, as well as the provision for overrides, is that the lossless component of the DCT coding exploits (among other things) statistical coherence between quantized coefficients whose coding positions are close to each other.

This will be clarified by the discussion of decoding and inversion of the DCT below.

H. Loop filter levels

L(6)	LoopFilterLevel
L(3)	SharpnessLevel

The meaning of these numbers will be explained in the loop filter discussion.

I. 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 after the discussion of coefficient decoding below.

J. The remaining frame header data occurs ONLY FOR INTERFRAMES

L(8)	ProbIntraPred = 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)	ProbLastPred = probability that an inter-predicted macroblock is predicted from the immediately previous frame, as opposed to the most recent golden 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 "YmodeTree" 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 "UvmodeTree" 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."

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 data, as well as the DCT coefficients from the second data partition, are known and will not change until the next frame.

10. MACROBLOCK FEATURES

Every macroblock may optionally override some of the default behaviors of the decoder. These overrides are called "features" and the prediction data for each (intra- or inter-coded) macroblock begins with a specification of up to four features, which are, in order:

1. A single 7-bit DCT dequantization index to be used for this macroblock, replacing all of the six indices used for the frame. This single index produces up to six different dequantization numbers via lookups into the same tables used for the frame-level dequantization.
2. A 6-bit loop filter level to be used for the top and left edges of this macroblock together with the interior subblock edges, again replacing the level selected for the frame as a whole.
3. A Bool which, if true, causes the partial update of the golden frame buffer with the reconstruction of this macroblock.
4. An 8-bit, two-part "pitch" specification for this macroblock, overriding the default raster placement of subblocks. The two parts of the pitch are, first, the "destination" (or "current-frame") pitch, which affects both intra-prediction and blitting of the reconstructed macroblock

into the current frame buffer and, second, the “prediction” (or “prior-frame”) pitch, which specifies the relation between subblocks and areas of the interframe prediction buffers..

Each feature takes a numeric argument which, on each frame, is restricted to four possibilities. These arguments are coded using the simple tree

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

combined with a 3-entry probability table.

The context for decoding features is provided by section B of the frame header. For each feature, in the same ordering as above, it contains:

1. A Flag which if 1 (0), enables (disables) the feature for this frame. The remaining fields occur if the feature is enabled.
2. The zero-probability “FeatProbNot” for the feature encoded as a P(8).
3. Three (optional) probabilities for decoding the numeric argument, each of the format F? P(8):255. These probabilities of course come in the same order as the entries in the decoding array and correspond to the indices 0,2,4 of the featureTree.
4. For the first, second, and fourth features, four optional possible values for the numeric argument, each having the format F? L(w):0. Here w is the width in bits of the numeric argument and takes the values w = 7,6, and 8, in accordance with the above.

The feature data at the beginning of each macroblock prediction record is then laid out as follows. For each ENABLED feature, in the above ordering, there is:a Bool(FeatProbNot) indicating whether the feature applies to this macroblock, followed, if true, by a selection of numeric argument using the tree and probability table described above.

Note that for the third feature (partial golden frame update), the choice of argument, while encoded, has no effect on decoding or reconstruction. This allows all four features to be decoded in exactly the same way with little increase in datarate: The information associated with this unused argument consumes three bits in the header and almost nothing in the macroblock records themselves.

Of the features, only the “pitch” requires further elaboration. Of all the aspects of VP7, the pitch has perhaps the highest confusion potential, so we discuss it at some length.

Only the high 5 bits of the 8-bit pitch value are meaningful. The top two bits select the current-frame “destination” pitch:

0. Normal raster-scan order: The (16 luma and 8 chroma) 4x4 subblocks tile the macroblock in raster order, the pixels in each subblock also occur in raster order.
1. FOUR: Chroma is normal. The jth luma subblock refers to the jth row of 16 Y pixels; j of course varies from 0 to 15.
2. X2: Both the luma and chroma are divided in the following “odd/even” fashion. Split the macroblock into two half-sized blocks, the first containing the “even” lines (0,2,4,6,8,10,12,14 of luma, 0,2,4,6 of chroma), the second containing the “odd” lines (1,3,5,7,9,11,13,15 of luma, 1,3,5,7 of chroma). The first eight Y, two U, and two V subblocks then tile the “even” half-macroblock in raster order, now restricted to the even lines, the pixels in each subblock occurring

in a similar “sub-raster” order. The “odd” half-macroblock is tiled by the last eight Y, two U, and two V subblocks in exactly the same way.

3. X4: The chroma is exactly as for X2. The luma portion of the macroblock is now divided into four 4x16 sections according to the position of the rows modulo 4 and consists of rows (0,4,8,12), (1,5,9,13), (2,6,10,14), and (3,7,11,15). The first four Y subblocks (0,1,2, and 3) tile the “mod 0” rows in sub-raster order, subblocks 4,5,6,and 7 do the same for the “mod 1” rows, subblocks 8 through 11 do the “mod 2” rows, and the last four subblocks 12 through 15 tile the “mod 3” rows 3, 7, 11, and 15.

The destination pitch specifies how this macroblock is “blitted” into the current-frame reconstruction buffer. It can also have an effect on intra prediction, as we will see below.

The modes “FOUR” and “X4” are NEVER used as destination pitches for macroblocks whose Y subblocks are independently predicted, i.e., those intra-macroblocks whose “Y mode” is “B_PRED” (this type of prediction is discussed in chapters 11, 12, and 16 below).

10.2 Inter-Prediction Pitch

Bits 5, 4, and 3 of the pitch value select the inter-prediction pitch, which affects interframe prediction only. Since the remainder of this chapter pertains only to this inter-prediction pitch, it perhaps should be read in conjunction with Chapter 18 (which treats the other details of inter-prediction).

The motion vector associated to such prediction, when added to the upper left corner position of the current macroblock, produces the upper-left corner of the prediction macroblock. The inter-prediction pitch affects the shape of, and positioning of subblocks within, the prediction macroblock.

As mentioned in Chapter 5, and to be detailed in Chapter 18, when a subblock is inter-predicted via a non-integral motion vector, “off the grid” pixels are “synthesized” by taking a weighted average of several pixels at small horizontal or vertical displacements from the position of each pixel to be synthesized. Because of this, aside from defining the shape of a referenced macroblock and its decomposition into subblocks, the inter-prediction pitch must also specify, for every position, which pixels are to be referenced by these horizontal and vertical displacements.

For the sake of definiteness, and in agreement with the process actually used, the “base” actual-pixel position of the interpolation is derived from a (possibly) fractional position by moving the position, if necessary, upward and/or leftward to the nearest integer grid position. We then need to say what it means to move horizontally or vertically from this base pixel position.

The handling of these displacements is relatively simple. In the first place, they are “translation-invariant”: The frame-buffer offsets specified by a displacement are the same regardless of the origin pixel at which they are based. Secondly, they are “linear”: If the vector “V” represents a single forward prediction step, the vector representing “n” prediction steps is simply n*V. This linearity also holds for “backward prediction steps” corresponding to $n < 0$.

Regardless of pitch, the horizontal prediction step is always “normal” ,that is, a single positive step simply moves right one pixel, which, in most frame buffer layouts, simply adds one to the address of the pixel.

It is the “vertical prediction step” (or “stride”) that varies with pitch. So, for each pitch, along with the macroblock shape and subblock decomposition,

We complete the description by also specifying this “stride,” namely, the vector representing a single forward vertical prediction step. For each pitch, there are two strides, one for luma (Y) and one for chroma (U and V).

We write strides (and all other vector displacements) in the form (R, C) where R = rows = vertical displacement within the frame buffer and C = columns = horizontal displacement within the frame buffer. For most frame buffer layouts, the conversion of an (R,C) to an address differential is effected by multiplying R by the size of a stored row in the frame buffer and adding C.

Values zero through three have the same meaning for the inter-prediction pitch as they do for the destination pitches described above. The associated strides are:

0.	Normal: Chroma stride and luma stride are both (1,0).
1.	FOUR: Chroma stride is (1,0), luma stride is (0,4).
2.	X2: Chroma stride and luma stride are both (2,0).
3.	X4: Chroma stride is (2,0), luma stride is (4,0).

The four additional values of inter-prediction pitch are as follows.

4. Plus1: Chroma is normal. The 16x16 luma prediction block has a “shear” of 45 degrees “southeast,” that is, the offsets of the sixteen 16-pixel long luma prediction rows relative to the upper left corner of the prediction block are: (0,0), (1,1), (2,2), ..., (14,14), and (15,15). The subblocks within the parallelogram-shaped prediction block occur in normal raster order, as do the pixels comprising the subblocks. Chroma stride = (1,0), luma stride = (1,1).
5. Minus1: Similar to Plus1 except now the shear is 45 degrees southwest, the relative line origins being (0,0), (1, -1), ..., (15,-15). Chroma stride = (1,0), luma stride = (1,-1).
6. X2plus1: Roughly, a combination of X2 and plus1. Chroma is exactly as in X2. The first four 4x4 luma blocks tile (in raster order) the four 16-pixel lines whose relative line origins are (0,0), (2,1), (4,2), and (6,3). Luma blocks 4 through 7 do the same for the lines based at (8,4), (10,5), (12,6), and (14,7). The last eight luma subblocks treat the odd lines similarly: blocks 8 through 11 tile the lines based at (1,0), (3,1), (5,2), and (7,3); blocks 12 thorough 15 tile the lines (9,4), (11,5), (13,6), and (15,7). Chroma stride = (2,0), luma stride = (2,1).
7. X2minus1: Similar to X2plus1 with the reverse shear, simply reversing the horizontal offsets used by X2plus1, that is, Y subblocks 0 through 3 correspond to lines (0,0), (2, -1), (4, -2), and (6, -3) and so on. Chroma stride = (2,0), luma stride = (2,-1).

It should be noted that the stride we have defined is, in every case, consistent with the adjacency relationships between pixels in the subblocks themselves. Put more explicitly, for all 8 pitches, and for each of the 16 luma (or 8 chroma) 4x4 subblocks, writing p(i,j) for the pixel in the ith row and jth column of the subblock, the frame-buffer offset between p(i,j) and p(i,j+1) is always (0,1) and the frame-buffer offset between p(i,j) and p(i+1,j) is always the luma (or chroma) stride.

This simplifies the inter-prediction process. The calculation of subsampled pixel values required by that prediction is, as will be further detailed in Chapter 18, the application of two one-dimensional “convolution” filters centered at each of 16 positions associated to the subblock (The case of whole pixel displacements and simple copying of pixels can be considered a degenerate special case of interpolation). Once we have calculated the origin of the subblock (based on the motion vector and relative position of the subblock), the relative addressing procedures involved in: (1) Moving the origins of these filters in accordance with the positions of subpixels to be synthesized, and (2) Referencing the actual prediction image pixels to be filtered, are identical. All we need to “know” in either case is what it means to move one pixel forward by one “horizontal prediction step” ((0,1) in the frame buffer) or one “vertical prediction step” (the stride).

In conclusion, we remark that, for chroma, the same 8x8 prediction block is used regardless of pitch, though whether the subdivision of the block is “normal” or “odd/even” is controlled by the pitch. For the

luma, the first four modes all use the usual 16x16 prediction block but divide it into subblocks in different ways. The last four inter-prediction modes use luma pixels outside the normal 16x16 prediction block.

We reiterate that the inter-prediction pitch has NOTHING TO DO with the relative positioning of subblocks (or pixels within subblocks) for the current frame being constructed; only the "normal" and "odd/even" (X2) modes of subdivision are supported for reconstruction and intra-prediction. Conversely, the current-frame destination pitch applied to reconstruction and intra-prediction has NOTHING TO DO with the inter-prediction pitch applied to earlier frames.

The decoding of macroblock feature records, together with the parsing of intra-prediction modes (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 (necessarily intra) prediction mode used for this macroblock.

11.1 Luma Modes

First comes the luma specification of type "intraMBmode," coded using the "kfYmodeTree," as described in the general discussion of tree coding above and repeated here for convenience:

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

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

   numUVmodes = B_PRED,  /* first four modes apply to chroma */
   numYmodes   /* all modes apply to luma */
}
intraMBmode;

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

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

   const Prob kfYmodeProb [numYmodes - 1] = { 145, 156, 163, 128};

   Ymode = (intraMBmode) TreedRead( d, kfYmodeTree, kfYmodeProb);

d is of course the BoolDecoder 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 is as
follows:

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

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

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

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

   numIntraBmodes
}
intraBmode;

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

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

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

11.2 Subblock Mode Contexts

The coding of subblock modes in key frames uses the modes already coded for the subblocks to the left and above the subblock to select the probability array used to decode the current subblock mode. There are a several caveats associated with this (our first instance of) contextual prediction:

1. The adjacency relationships between subblocks are based on the normal default raster placement of the subblocks and are not affected by any pitch overrides that may be in effect for this macroblock.
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 of us. 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. Since any sensible implementation of subblock-mode decoding will maintain a buffer of already-calculated modes, it may find it convenient to pad this buffer with an "invisible" row above (and column to the left of) the actual predictors set to the constant value B_DC_PRED.
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_LD_PRED (not B_TM_PRED, as one might guess).
5. Although we discuss interframe modes later, we remark here that, while interframes do use all the intra coding modes described here and below, the subblock modes in an interframe are coded using a single constant probability array that does not depend on any context.

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

```
const Prob kfBmodeProb [numIntraBmodes] [numIntraBmodes] [numIntraBmodes-1];
```

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

```
Bmode = (intraBmode) TreedRead( d, BmodeTree, kfBmodeProb [A] [L]);
```

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

The contents of the "kfBmodeProb" array are given at the end of this section.

11.3 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 YVmodeTree described in the tree-coding discussion above, again repeated here for convenience:

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

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

```
const Prob kfUVmodeProb [numUVmodes - 1] = { ??, ??, ??};

UVmode = (intraMBmode) TreedRead( d, UVmodeTree, kfUVmodeProb);
```

This completes the description of macroblock prediction coding for key frames. As will be discussed below, the coding of intra modes within interframes is similar, but not identical, to that described above.

We next describe the prediction buffer calculations selected by these modes but, before leaving the topic of coding, we make one remark.

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 here (and in the reference code) for prediction modes and, indeed, for all tree-coded data in VP7.

11.4 Subblock Mode Probability Table

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

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

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

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

12. INTRA PREDICTION PROCESS

The intra prediction process is invoked for:

- Macroblocks within an Intra frame;
- Macroblocks within an Inter frame with *refFrame* set to INTRA_FRAME.

There are three types of Intra Prediction process:

- 4x4 Intra Prediction
- 16x16 Intra Prediction
- 8x8 Intra Prediction

The 4x4 Intra Prediction and 16x16 Intra Prediction are used for luma samples only, 8x8 Intra Prediction is used for chroma samples only.

12.1 and 12.2 describe the two types of intra prediction process for luma samples, and 12.3 describe the intra prediction process fro chroma samples.

12.1 4x4 Intra Prediction Process

This process is invoked for each 4x4 block within a macroblock when the *mode* of the macroblock is set to B_PRED.

As the prediction of 4x4 Intra block samples uses the reconstructed samples to the left and above, the 4x4 Intra prediction process is interleaved with the 4x4 reconstruction process, i.e. after each 4x4 intra prediction process, the reconstruction process for the same 4x4 block is invoked to produce the

reconstructed samples, which will be used for the prediction process if 4x4 blocks to the right and/or below. It should be noted here that the reconstructed samples used are prior to the application of loop filter.

Inputs to this process are the reconstructed samples prior to the loop filter process from adjacent blocks and the Block Prediction Mode for the current 4x4 luma block. Figure 12.1 illustrates the 13 reconstructed pixel samples that can be used in this process depending on the *Block Prediction Mode*: TL, A[0], A[1], …, A[7], L[0], L[1], L[2], and L[3].

TL	*A[0]*	*A[1]*	*A[2]*	*A[3]*	*A[4]*	*A[5]*	*A[6]*	*A[7]*
L[0]	*x*	*x*	*x*	*x*				
L[1]	*x*	*x*	*x*	*x*				
L[2]	*x*	*x*		*x*	*x*			
L[3]	*x*	*x*		*x*	*x*			

Figure 12.1 Input Pixel Samples in Block Intra Prediction Process

Where the 4x4 block of "x" represents the current 4x4 block of luma samples the prediction block is calculated for.

For any macroblock with *Prediction Mode* set to B_PRED, A [4], …, A[7] are not reconstructed yet for 3 out of the 4 rightmost blocks within the macroblock since the Macroblock to the right will be decoded and reconstructed after the current macroblock. In such situation, decoder shall use A[4],…, A[7] from the closest above block in which A[4], …, A[7] have already been reconstructed.

Outputs of this process are 4x4 prediction samples for the 4x4 luma block.

For convenience, in the rest of this sub-section, *Predictor[i], i=0, 1, …, 15* will be used to represent the prediction output samples for a 4x4 block. The index *i* increments in raster order, i.e. 0, 1, 2 and 3 for the first row, 4, 5, 6 and 7 for the second row, and so on.

(It should be noted that when the Prediction Buffer is allocated for a 16x16 macroblock, the distance between sample 0 and sample 4 inside such buffer is 16.)

Depending on the *Block Prediction Mode*, one of the following 10 block prediction modes specified in 12.1.1 to 12.1.10 shall be used.

12.1.1 B_DC_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_DC_PRED.

The values of prediction samples are derived by:

$$expectedDC = (A[0] + A[1] + A[2] + A[3] + L[0] + L[1] + L[2] + L[3] + 4) >> 3$$

$$Predictor[i] = expectedDC,\ i = 0, \ldots, 15$$

12.1.2 B_TM_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_TM_PRED.

The values of prediction samples are derived by:

$$Predictor[m*4 + i] = clamp255(L[m] + A[i] - TL),\ i, m = 0, \ldots, 3$$

12.1.3 B_VE_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_VE_PRED.

The values of prediction samples are derived by:

*Predictor [m*4+0] = (TL + A[0]*2 + A[1]+2)>>2, m = 0, …, 3*

*Predictor [m*4+1] = (A[0] + A[1]*2 + A [2]+2)>>2, m = 0, …, 3*

*Predictor [m*4+2] = (A[1] + A[2]*2 + A[3]+2)>>2, m = 0, …, 3*

*Predictor [m*4+3] = (A[2] + A[3]*2 + A [4]+2)>>2, m = 0, …, 3*

12.1.4 B_HE_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_HE_PRED.

The values of prediction samples are derived by:

*Predictor [m] = (TL + L[0]*2 + L[1]+2)>>2, m = 0, …, 3*

*Predictor [4 +m] = (L[0] + L[1]*2 + L[2]+2)>>2, m = 0, …, 3*

*Predictor [8 +m] = (L[1] + L[2]*2 + L[3]+2)>>2, m = 0, …, 3*

*Predictor [12+m] = (L[2] + L[3]*2 + L[3]+2)>>2, m = 0, …, 3*

12.1.5 B_LD_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_LD_PRED.

The values of prediction samples are derived by:

*Predictor [0] = (A[0] + A[1] * 2 + A[2] +2)>>2;*
Predictor [1] =
*Predictor [4] = (A[1] + A[2] * 2 + A[3] +2)>>2;*
Predictor [2] =
Predictor [5] =
*Predictor [8] = (A[2] + A[3] * 2 + A[4] +2)>>2;*
Predictor [3] =
Predictor [6] =
Predictor [9] =
*Predictor [12] = (A[3] + A[4] * 2 + A[5] +2)>>2;*
Predictor [7] =
Predictor [10] =
*Predictor [13] = (A[4] + A[5] * 2 + A[6] +2)>>2;*
Predictor [11] =
*Predictor [14] = (A[5] + A[6] * 2 + A[7] +2)>>2;*
*Predictor [15] = (A[6] + A[7] * 2 + A[7] +2)>>2;*

12.1.6 B_RD_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_RD_PRED.

The values of prediction samples are derived by:

*Predictor [12] = (L[3] + L[2] * 2 + L[1] + 2)>>2;*
Predictor [13] =
*Predictor [8] = (L[2] + L[1] * 2 + L[0] + 2)>>2;*
Predictor [14] =
Predictor [9] =
*Predictor [4] = (L[1] + L[0] * 2 + TL + 2)>>2;*
Predictor [15] =
Predictor [10] =
Predictor [5] =

*Predictor [0] = (L[0] + TL * 2 + A[0] + 2)>>2;*
Predictor [11] =
Predictor [6] =
*Predictor [1] = (TL + A[0] * 2 + A[1] + 2)>>2;*
Predictor [7] =
*Predictor [2] = (A[0] + A[1] * 2 + A[2] + 2)>>2;*
*Predictor [3] = (A[1] + A[2] * 2 + A[3] + 2)>>2;*

12.1.7 B_VR_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_VR_PRED.

The values of prediction samples are derived by:

*Predictor [12] = (L[2] + L[1] *2 + L[0] +2)>>2;*
*Predictor [8] = (L[1] + L[0] *2 + TL +2)>>2;*
Predictor [13] =
Predictor [4] = (L[0] + TL + A[0] +2)>>2;
Predictor [9] =
Predictor [0] = (TL + A[0] + 1)>>1;
Predictor [14] =
*Predictor [5] = (TL [4] + A [0] *2 + A[1] +2)>>2;*
Predictor [10] =
Predictor [1] = (A[0] + A[1] + 1)>>1;
Predictor [15] =
*Predictor [6] = (A[0] + A[1] *2 + A[2] +2)>>2;*
Predictor [11] =
Predictor [2] = (A[1] + A[2] + 1)>>1;
*Predictor [7] = (A[1] + A[2] *2 + A[3] +2)>>2;*
Predictor [3] = (A[2] + A[3] + 1)>>1;

12.1.8 B_VL_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_VL_PRED.

The values of prediction samples are derived by:

Predictor [0] = (A[0] + A[1] + 1)>>1;
*Predictor [4] = (A[0] + A[1] *2 + A[2] +2)>>2;*
Predictor [8] =
Predictor [1] = (A[1] + A[2] + 1)>>1;
Predictor [5] =
*Predictor [12] = (A[1] + A[2] *2 + A[3] +2)>>2;*
Predictor [9] =
Predictor [2] = (A[2] + A[3] + 1)>>1;
Predictor [13] =
*Predictor [6] = (A[2] + A[3] *2 + A[4] +2)>>2;*
Predictor [3] =
Predictor [10] = (A[3] + A[4] + 1)>>1;
Predictor [7] =
*Predictor [14] = (A[3] + A[4] *2 + A[5] +2)>>2;*
*Predictor [11] = (A[4] + A[5] *2 + A[6] +2)>>2;*
*Predictor [15] = (A[5] + A[6] *2 + A[7] +2)>>2;*

12.1.9 B_HD_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_HD_PRED.

The values of prediction samples are derived by:

Predictor [12] = (L[3] + L[2] + 1)>>1;
*Predictor [13] = (L[3] + L[2] *2 + L[1] +2)>>2;*
Predictor [8] =
Predictor [14] = (L[2] + L[1] + 1)>>1;
Predictor [9] =
*Predictor [15] = (L[2] + L[1] *2 + L[0] +2)>>2;*
Predictor [10] =
Predictor [4] = (L[1] + L[0] + 1)>>1;
Predictor [11] =
*Predictor [5] = (L[1] + L[0] *2 + TL +2)>>2;*
Predictor [6] =
Predictor [0] = (L[0] + TL + 1)>>1;
Predictor [7] =
*Predictor [1] = (L[0] + TL *2 + A[0] +2)>>2;*
*Predictor [2] = (TL + A[0] *2 + A[1] +2)>>2;*
*Predictor [3] = (A[0] + A[1] *2 + A[2] +2)>>2;;*

12.1.10 B_HU_PRED

This intra 4x4 prediction process is invoked when *Block Prediction Mode* is set to B_HU_PRED.

The values of prediction samples are derived by:

Predictor [0] = (L[0] + L[1] + 1)>>1;
*Predictor [1] = (L[0] + L[1] *2 + L[2] +2)>>2;*
Predictor [2] =
Predictor [4] = (L[1] + L[2] + 1)>>1;
Predictor [3] =
*Predictor [5] = (L[1] + L[2] *2 + L[3] +2)>>2;*
Predictor [6] =
Predictor [8] = (L[2] + L[3] + 1)>>1;
Predictor [7] =
*Predictor [9] = (L[2] + L[3] *2 + L[3] +2)>>2;*
Predictor [10] =
Predictor [11] =
Predictor [12] =
Predictor [13] =
Predictor [14] =
Predictor [15] = L[3];

12.2 16x16 Intra Prediction Process

This process is invoked when the *mode* of a macroblock is set to one of the 16x16 intra prediction modes: DC_PRED, V_PRED, H_PRED, TM_PRED.

Inputs to this process are the reconstructed samples prior to the loop filter process from adjacent macroblocks and *mode*. Figure 12.2 illustrates the reconstructed pixel samples that shall be used in this process: TL, A[0], A[1], …, A[15], L[0], L[1], …, L[15]. For macroblocks on the left and top of the image boundary, these samples can be outside of the image boundary. In such case, the unavailable samples shall be replaced by the value of 128.

Outputs of this process are 16x16 luma prediction samples for the macroblock.

For convenience, in the rest of this sub-section, $P_M[i],\ i=0, 1, \ldots, 255$ will be used to represent the prediction output samples. The index i increments in raster order, i.e. 0, 1,…,15 are for the first row; 16, 17, …,31 are for the second row; and so on.

$$\begin{array}{ccccc} TL & A[0] & A[1] & \cdots & A[15] \\ L[0] & P_M[0] & P_M[1] & \cdots & P_M[15] \\ L[1] & P_M[16] & P_M[17] & \cdots & P_M[31] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L[15] & P_M[240] & P_M[241] & \cdots & P_M[255] \end{array}$$

Figure 12.2 Input and output of the 16x16 Intra Prediction Process

Depending on a macroblock's *mode*, one of the following 4 prediction modes specified in 12.2.1 to 12.2.4 shall be used.

12.2.1 DC_PRED

This intra 16x16 prediction process is invoked when *mode* is set to DC_PRED.

The process first calculates the average value *expectedDC* of available reconstructed samples among A[0], A[1], …, A[15] and L[0], L[1], …, L[15], i.e., when all input samples are available:

$$expectedDC = \left(\sum_{i=0}^{15} L[i] + \sum_{i=0}^{15} A[i] + 16 \right) >> 5$$

If only the input samples from the Left are available, then

$$expectedDC = \left(\sum_{i=0}^{15} L[i] + 8 \right) >> 4$$

Or only the input samples from the Above are available, then

$$expectedDC = \left(\sum_{i=0}^{15} A[i] + 8 \right) >> 4$$

Then all the prediction samples are assigned with the value of *expectedDC*, i.e.

$$P_M[i] = expectedDC \quad (i = 0,1,\ldots,255)$$

12.2.2 V_PRED

This intra 16x16 prediction process is invoked when *mode* is set to V_PRED.

The values of prediction samples are derived by:

$$P_M[m*16+n] = A[n],\ (m,n=0,1,..,15)$$

12.2.3 H_PRED

This intra 16x16 prediction process is invoked when *mode* is set to H_PRED.

The values of prediction samples are derived by:

$$P_M[m*16+n] = L[m],\ (m,n=0,1,..,15)$$

12.2.4 TM_PRED

This intra 16x16 prediction process is invoked when *mode* is set to TM_PRED.

The values of prediction samples are derived by:

$$P_M[m*16+n] = clamp255(L[m]+A[n]-TL),\ (m,n=,1,..,15)$$

12.3 8x8 Intra Prediction Process

This process is invoked when the *UVmode* of a macroblock is set to one of the 8x8 intra prediction modes: DC_PRED, V_PRED, H_PRED, TM_PRED.

This process is used for the chroma samples of a macroblock only. The same process is used for both U and V chroma planes except the reconstructed samples are from each individual chroma plane. For convenience, the process of only one chroma plane is described here. In the rest of this sub-section, P_M *[i], i=0, 1, …, 63* will be used to represent the prediction output samples in one chroma plane. The index *i* increments in raster order, i.e. 0, 1,…,7 are for the first row; 8, 9, …,15 are for the second row; and so on.

Inputs to this process are the reconstructed samples prior to the loop filter process from adjacent macroblocks and *UVmode*. Figure 12.3 illustrates the reconstructed pixel samples that shall be used in this process: TL, A[0], A[1], …, A[7], L[0], L[1], …, L[7]. For macroblocks on the left and top of the image boundary, these samples can be outside of the image boundary. In such case, the unavailable samples shall be replaced by the value of 128.

Outputs of this process are two planes of 8x8 chrome prediction samples for the macroblock.

$$\begin{array}{ccccc} TL & A[0] & A[1] & \cdots & A[7] \\ L[0] & P_M[0] & P_M[1] & \cdots & P_M[7] \\ L[1] & P_M[8] & P_M[9] & \cdots & P_M[15] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L[7] & P_M[56] & P_M[57] & \cdots & P_M[63] \end{array}$$

Figure 12.3 Input and output of the 16x16 Intra Prediction Process

Depending on a macroblock's *UVmode*, one of the following 4 prediction modes specified in 12.3.1 to 12.3.4 shall be used.

12.3.1 DC_PRED

This intra 8x8 prediction process is invoked for both chroma planes when *UVmode* is set to DC_PRED.

The process first calculates the average value *expectedDC* of available reconstructed samples among A[0], A[1], …, A[7] and L[0], L[1], …, L[7], i.e., when all input samples are available:

$$expectedDC = \left(\sum_{i=0}^{7} L[i] + \sum_{i=0}^{7} A[i] + 8 \right) >> 4$$

If only the input samples from the Left are available, then

$$expectedDC = \left(\sum_{i=0}^{7} L[i] + 4 \right) >> 3$$

Or only the input samples from the Above are available, then

$$expectedDC = \left(\sum_{i=0}^{7} A[i] + 4 \right) >> 3$$

Then all the prediction samples are assigned with the value of *expectedDC*, i.e.

$$P_M[i] = expectedDC \quad (i = 0,1,\ldots,63)$$

12.3.2 V_PRED

This intra 8x8 prediction process is invoked for both chroma planes when *UVmode* is set to V_PRED.

The values of prediction samples are derived by:

$$P_M[m*8+n] = A[n], (m,n=0,1,..,7)$$

12.3.3 H_PRED

This intra 8x8 prediction process is invoked for both chroma planes when *UVmode* is set to H_PRED.

The values of prediction samples are derived by:

$$P_M[m*8+n] = L[m], (m,n=0,1,..,7)$$

12.3.4 TM_PRED

This intra 8x8 prediction process is invoked for both chroma planes when *UVmode* is set to TM_PRED.

The values of prediction samples are derived by:

$$P_M[m*8+n] = clamp255(L[m]+A[n]-TL), (m,n=0,1,..,7)$$

13. DCT COEFFICIENT DECODING

The second data partition consists of (a rather elaborate encoding of) the quantized DCT coefficients of the residue signal. As discussed in the decoding overview, 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.

VP7 works exclusively with 4x4 DCTs, each of which applies to one of the 4x4 subblocks of a macroblock. The ordering of macroblocks in the second “residue” partition is of course the same raster-scan as is used in the first “prediction” partition.

The structure of each macroblock’s residue record is almost independent of the prediction, the only dependence being that, except for intra-coded macroblocks whose Y mode is “B_PRED” (that is, whose Y subblocks are independently predicted), the record begins with the “Y2” component of the residue; the B_PREDicted macroblocks omit this DCT, instead specifying the 0th DCT coefficient of each of the 16 Y subblocks as part of its DCT.

If the “Y2” block is present (absent), the remainder (totality) of the residue record consists of 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 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.

The ordering of the coefficients can be variable and is as described in the frame header above. All chroma (and Y2, if present) DCTs begin at coefficient zero (always DC). The 16 luma DCTs begin at coefficient 1 (whose row, column depend on the coefficient ordering) if Y2 is present and begin at coefficient 0 if Y2 is absent.

13.1 Coding Of Individual Coefficient Values

All tokens 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.

Nonzero values (of either type) are then followed by a Flag indicating the sign of the coded value (negative if 1, positive if 0).

Here are the tokens and decoding tree.

```
typedef enum
{
   DCT_0,      /* value 0 */
   DCT_1,      /* 1 */
   DCT_2,      /* 2 */
   DCT_3,      /* 3 */
   DCT_4,      /* 4 */
   DCT_cat1,   /* range 5 - 6  (size 2) */
   DCT_cat2,   /* 7 - 10   (4) */
   DCT_cat3,   /* 11 - 18  (8) */
   DCT_cat4,   /* 19 - 34  (16) */
   DCT_cat5,   /* 35 - 66  (32) */
   DCT_cat6,   /* 67 - 2048  (1982) */
   DCT_eob, /* end of block */

   numDCT_tokens  /* 12 */
}
DCT_token;

const TreeIndex CoefTree [2 * (numDCT_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" */
};
```

The tokens DCT_cat1 ... DCT_cat6 specify ranges of unsigned values, the value within the range being formed by adding an unsigned offset (whose width is 1, 2, 3, 4, 5, or 11 bits, respectively) to the base of the range, using the following algorithm and fixed probability tables.

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

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

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

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

13.2 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 by section “I” of the frame header. The full (non-constant) probability table is laid out as follows.

```
   Prob coefProbs [4] [8] [3] [numDCT_tokens-1];
```

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

0. Y beginning at coefficient 1 (i.e., Y after Y2)

1. Y2

2. U or V

3. Y beginning at coefficient 0 (i.e., Y in the absence of Y2).

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

```
   CoefBands [c]
```

where

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

is a fixed mapping of position to “band.”

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

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 nonzero 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 zero if the last coefficient was a zero, one if it was plus or minus one, and two 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 of the use of end-of-block, 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 OK 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 VP7, 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 zero.

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 set to zero, in compliance with the definition of “non-existent” prediction.

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

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

13.3 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; 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 coefProbs array, there occurs a fixed-probability bool indicating whether or not the corresponding probability should be updated. If the bool is true, there follows a P(8) replacing that probability. Note that updates are cumulative, that is, a probability updated on one frame is in effect for all ensuing frames until the next key frame, or until the probability is explicitly updated by another frame.

The algorithm to effect the foregoing is simple:

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

      if( readBool( d, coefUpdateProbs [i] [j] [k] [t]))
         coefProbs [i] [j] [k] [t] = readLiteral( d, 8);
```

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

The (constant) update probability array may be found in the reference decoder file "coefUpdateProbs.c."

13.4 Default Token Probability Table

The default token probabilities are as follows.

```
const Prob defaultCoefProbs [4] [8] [3] [numDCT_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}
  }
 }
};
```

14. DCT INVERSION AND MACROBLOCK RECONSTRUCTION

After decoding of the DCT 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 the corresponding dequantization tables (found in the reference decoder file "quant_common.c") using the single index supplied by the override. Otherwise, the frame-level dequantization factors (as described in section "C" of the frame header) are used. In either case, the multiplies are computed and stored using 16-bit signed integers.

14.1 DC Prediction

Next, a DC prediction offset is optionally added to the zeroth Y2 coefficient (that is, the DC level associated to the entire 16x16 luma block) of an inter-predicted macroblock. VP7 decides whether or not to do this based on how well previously-coded "candidate" macroblocks "match" incoming macroblocks. The "candidates," for purposes of both "tracking" and prediction itself, are precisely the inter-coded macroblocks. The candidates are segregated according to their reference frame (golden or previous). The two classes are tracked and predicted with regard only to themselves, that is, macroblocks whose reference frame is golden (previous) have no impact on macroblocks whose reference frame is previous (golden), as far as DC prediction is concerned.

Each class is tracked via the maintenance of two values. The first is called the "prediction value" (abbreviated "PV") and is the last Y2 DC coefficient actually used by the previous candidate macroblock (whether offset via prediction or not). The second is a "match count." If this count exceeds three, the PV is added to the dequantized coefficient "C." The match count is incremented if (and only if) the (possibly adjusted) C equals the PV (determined by previous candidates) and they are nonzero. If either C or PV is zero, or if their signs disagree, the match count is reset to zero. If C and PV are both nonzero, have the same sign, but are not identical, the match count is left alone.

Regardless of the match count (which now impacts only future candidates), the (possibly adjusted) value "C" is both the actual zeroth Y2 coefficient used in the reconstruction of the current macroblock and the prediction value in effect for the next candidate.

The prediction value and match count are maintained across interframes and are both reset to zero every key frame. In the reference decoder, the dequantization and DC prediction are invoked in "dequantize.c," the DC prediction itself is in "predictdc.c," and the quantization indices are handled in "decodframe.c."

14.2 Inverse DCT Transform

Next comes the inversion of the DCTs. If the Y2 residue block exists (i.e., the macroblock mode is not B_PRED), it is inverted first and the element of the result at row i, column j is used as the zeroth coefficient of the Y subblock at position (i, j), that is, the Y subblock whose index is (i * 4) + j. As discussed above, if the mode is B_PRED, the zeroth 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 and also have nothing to do with any pitch override that may be in effect for this macroblock; 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 result pixels.

All of the DCT inversions (including Y2) are computed in exactly the same way. In principle, VP7 uses the “DCT-II” as its forward transform. The basis function “b” at row i, column j, a function of vertical position y and horizontal position x, is a product of independent one-dimensional cosines:

```
b( i, j; y, x) = c( i; y) * c( j; x);
```

i, j, y, and x all take the values 0, 1, 2, or 3. The cosine functions are

```
c( i; y) = N(i) * cos( pi * (i/4) * (y + 1/2) );
```

here N(i) is a normalization factor which equals 1/2 for i = 0 and the square root of 1/2 for i nonzero.

The 4x4 forward DCT f(i, j) is then given by taking the inner (or dot) product of the original (pixel sampled) function s with the (i,j)th basis function

```
f( i, j) = Sum{ s( x, y) * c( i; y) * c( j; x) : 0 <= x,y <= 3}.
```

So defined, our DCT is unitary, that is, over a 4x4 block, the sum of the squares of the coefficients equals the sum of the squares of the original samples. The DCT is inverted by taking a sum over all the basis functions, each weighted by the corresponding coefficient

```
ss( x, y) = Sum{ f( i, j) * c( i; y) * c( j; x) : 0 <= i,j <= 3}.
```

As for any (real) unitary transform, the inverse is formed by taking the transpose of the forward matrix, which in our case amounts to exchanging the spatial and frequency variables.

Because the basis functions are products of one-dimensional functions, both the forward and inverse DCTs may be computed one variable at a time. For example, computing the inverted result “ss” by summing first in j and then in i may be thought of as taking the vertical inverse transform of the horizontal inverse transform (and this is the order in which VP7 performs the inversion).

VP7 of course uses a finite-precision approximation to the above. Also, the forward DCT used by VP7 has double the normalization of the standard unitary transform, that is, every dequantized coefficient has roughly double the size of the corresponding unitary coefficient. 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.

14.3 Implementation of DCT Inversion

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.

```
/* m [i] [y] is approximately pow( 2, 15.5) * c( i; y). */

const int16 m [4] [4] =
{
   23170,  23170,  23170,  23170,   /* c( 0; y) = constant */
   30274,  12540, -12540, -30274,   /* c( 1; y) = a half wave */
   23170, -23170, -23170,  23170,   /* c( 2; y) = a full wave */
   12540, -30274,  30274, -12540 /* c( 3; y) = a wave and a half */
};

/* in = dequantized coefficients, out = result.
   Each time we apply m, we multiply the rms norm by (roughly) pow( 2, 15.5).
   After the horizontal scaling, we've multiplied by pow( 2, 1.5). After the
   vertical (final) scaling, we've multiplied by pow( 2, 1.5 + 15.5 - 18) =
   pow( 2, -1) = 1/2, compensating for the    double forward normalization
   used by VP7. */

void idct( const int16 in[4][4], int16 out[4][4])
{
   int16 b [4] [4];           /* temporary result buffer */

   {                        /* start with horizontal inverse */
      int i = 0;  do          /* each row is treated identically */
      {
         int x = 0;  do       /* each point within row is */
         {
            int32 a = 0;      /* a dot product */
            int j = 0;  do    /* summed over 4 frequencies */
            {
               a += in[i][j] * m[j][x];
            }
            while( ++j < 4);

            b[i][x] = (a + (1<<13)) >> 14; /* scale by  pow( 2, -14) */
         }
         while( ++x < 4);     /* next point within row */
      }
      while( ++i < 4);        /* next row */
   }

   {                        /* finish with vertical inverse */
      int x = 0;  do          /* each column is treated identically */
      {
         int y = 0;  do       /* each point within column is */
         {
            int32 a = 0;      /* a dot product */
            int i = 0;  do    /* summed over 4 frequencies */
            {
               a += b[i][x] * m[i][y];
            }
            while( ++i < 4);
```

```
            out[x][y] = (a + (1<<17)) >> 18; /* scale by  pow( 2, -18) */
         }
         while( ++y < 4);      /* next point within column */
      }
      while( ++x < 4);         /* next column */
   }
}
```

The reference decoder DCT inversion may be found in the files “invtrans.c” and “idct.c.”

14.4 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.

The destination pitch, of which there are two (supported) cases, must be respected here. If the pitch is normal, the subblocks are “blitted” into the reconstructed frame buffer in normal raster-scan order. If the pitch is “X2,” the subblocks are blitted according to the “odd/even” macroblock subdivision described in Chapter 10.

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

As previously mentioned, the purpose of the loop filter is to smooth block discontinuities in the reconstruction buffer that is used as the prediction reference for decoding subsequent frames. Such smoothing helps to prevent the propagation of these discontinuities in subsequent frames.

Note that loop filtering is quite distinct from post processing, which affects only the displayed image and not the prediction buffer. Whereas post processing may be arbitrarily switched on and off by the decoder without affecting the integrity of the bitstream, any loop filtering done by the decoder must exactly match that done by the encoder. Hence any alterations or variations to the level or type of loop filtering must be determined by the encoder and fully specified in the bitstream.

VP7 supports two types of loop filtering (see 9.F) defined as “normal” and “simple.” The algorithms for these are described briefly below and unless otherwise stated are detailed in the code module “loopfilter.c.”

In addition there are two further parameters (see 9.H) that influence the behavior of the loop filter. These are “LoopFilterLevel” and “SharpnessLevel.”

In most situations “normal” loop filtering will be the preferred option as this generally results in superior subjective and objective quality. The simple loop filter option is included for use in situations where it is known in advance that decoder performance is likely to be more important than quality. In

cases where decoder performance is known to be extremely critical it is also possible to disable loop filtering altogether by setting the LoopFilterLevel to 0.

Note that a compliant decoder MUST capable of applying both “normal” or “simple” loop filtering and must fully implement the functionality provided for by the LoopFilterLevel and SharpnessLevel parameters, as the selection of filter type, level and sharpness are made by the encoder.

15.1 The Simple Loop Filter

The simple loop filter is designed with SIMD instructions in mind and can be implemented exclusively using “char” (signed 8 bit) arithmetic. Further, the ‘C’ code is designed to map easily to a SIMD implementation rather than to be optimal in its own right.

It is implemented by the following functions, which can be found in “loopfilter.c.”

```
LoopFilterSimpleVerticalEdgeC()
LoopFilterSimpleHorizontalEdgeC()
SimpleFilterMask()
SimpleFilter()
```

When using the “simple” loop filter, both macro-block (16x16 in Y, 8x8 in U & V) and block (4x4) boundaries are treated in the same way.

```
                    Block Boundary
                          |
              p1    p0    |     q0    q1
```

If four adjacent pixels (Luma or Chroma) p1, p0, q0, q1 lie along a vertical or horizontal line such that there is a block boundary between p0 and q0 then the simple loop filter can be summarized as follows:

```
FiltVal = (p1 - 3*(p0 - q0) - q1) >> 3;
q0 -= FiltVal;
p0 += FiltVal;
```

However, the reality is complicated by clamping and rounding issues in respect of which the reader is referred to the code. Further, masking is applied such that the filter is disabled and has no effect if the absolute value of (p0 – q0) exceeds a threshold value given by the value of “LoopFilterLevel” (see 9.H).

15.2 The Normal Loop Filter

The normal loop filter is also designed with SIMD instructions in mind and can in the main be implemented using “char” (signed 8 bit) arithmetic. Note that the ‘C’ code is designed to map easily to a SIMD implementation rather than to be optimal in its own right.

Unlike the simple loop filter, the filtering on block (4x4) boundaries is different to that applied to macro-block (16x16Y, 8x8U&V) boundaries.

It is implemented principally by the following functions, which can be found in loopfilter.c

```
/* Block filter functions */
LoopFilterHorizontalEdgeC()
LoopFilterVerticalEdgeC()
HEVMask()
FilterMask()
Filter()
```

```
/* Macro-block filter functions */
MBLoopFilterHorizontalEdgeC()
MBLoopFilterVerticalEdgeC()
HEVMask()
FilterMask()
MBFilter()
```

15.3 4x4 Pixel Block Boundary Filter

Consider four adjacent pixels (Luma or Chroma) p1, p0, q0, q1 lie along a vertical or horizontal line such that there is a block boundary between p0 and q0.

```
                          Block Boundary
                                |
                    p1    p0    |     q0    q1
```

The "normal" block filter is based upon the same underlying filter as that used for all boundaries in the simple filter. The basic method is extended by applying some additional constraints to the filtering of p0 and q0 and by allowing for adjustment to the values of p1 and q1. The method can be summarized as follows:

```
HighEdgeVariation = (abs(p1 – p0) > InnerThresh) ||
        (abs(q1 – q0) > InnerThresh)

// Restrict the use of p1 and q1 to cases where the difference between
// p0 and p1 or between q0 and q1 is above a threshold value
if (HighEdgeVariation)
    FiltVal = (p1 – q1);

FiltVal = (Filter + 3 * ( qs0 – ps0)) >> 3;
q0 –= FiltVal;
p0 += FiltVal;

// Apply adjustment to p1 and q1 as well if HighInnerDiff is false
if ( !HighEdgeVariation )
{
FiltVal += 1;
FiltVal = FiltVal >> 1;
q1 –= FiltVal;
p1 += FiltVal;
}
```

As with the simple loop filter, there are additional rounding, clamping constraints that are detailed in the code and a threshold given by the value of "LoopFilterLevel" (see 9.H), is applied such that where the value of abs (p0 – q0) exceeds the threshold no filtering is applied and p1, p0, q0 or q1 are left unchanged.

However, in the case of the "normal" filter there are additional constraints relating to the relative values of points further from the boundary ((p3-p2), (p2-p1), (p1-p0), (q1-q0), (q2-q1) and (q3-q2)) (see `FilterMask()`). These are tested against a second smaller threshold whose value (calculated in the function `ONYX_LoopFilterFrame()`) is a function of both "LoopFilterLevel" and "SharpnessLevel."

It is worth noting that in the code the thresholding is actually implemented via a masking mechanism (see `FilterMask() and HEVMask()`).

15.4 Macro Block Boundary Filter

When the normal loop filter is selected this filter is applied to the macro block boundaries (16x16 pixel boundaries in Y and 8x8 boundaries in U & V). Unlike the block boundary filter, which can only alter pixels up to a distance of 2 pixels either side of the boundary, the macro block filter has a slightly larger maximum "range" of +/- 3 pixels.
The filter is implemented in `MBFilter` () and can be summarized as follows:

```
HighEdgeVariation (abs(p1 – p0) > InnerThresh) ||
      (abs(q1 – q0) > InnerThresh)


// Where edge variation is high filter p0 and q0 as if it
// were a block boundary.
if (HighEdgeVariation)
   …

// Low edge variance case
else
{
FiltVal = (p1 – 3*(p0 – q0) – q1);

   u = (63 + (FiltVal * 27)) >> 7
   q0 –= u;
   p0 += u;

   u = (63 + (FiltVal * 18)) >> 7
   q2 –= u;
   p2 += u;

   u = (63 + (FiltVal * 9)) >> 7
   q2 –= u;
   p2 += u;
}
```

As with the simple filter and the normal block filter various rounding and clamping constraints are applied in respect of which the reader is referred to the code.

As with the normal block filter, the macro block filter is not applied if limiting threshold values for (p0-q0), (p3-p2), (p2-p1), (p1-p0), (q1-q0), (q2-q1) and (q3-q2) are exceeded. These thresholds are enforced in the code via masking operations.

16. INTERFRAME MACROBLOCK PREDICTION RECORDS

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

After the "feature" specification as described above (which is identical for keyframes and interframes), there comes a Bool (ProbIntraPred), which indicates inter-prediction when true and intra-prediction when false. ProbIntraPred is set by field "J" of the frame header.

16.1 Intra-Predicted Macroblocks

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

As discussed in the description of tree coding above, the "outer" Y mode here uses a different tree from that used for key frames, repeated here for convenience.

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

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

```
Prob YmodeProb [numYmodes – 1] = { 112, 86, 140, 37};

Prob UVmodeProb [numUVmodes – 1] = { 162, 101, 204};
```

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

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

```
const Prob BmodeProb [numIntraBmodes – 1] = { ?? };
```

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

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

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(ProbLastPred), selecting the reference frame (previous frame if 0, golden frame if 1) in effect for the entire macroblock. The probability "ProbLastPred" is 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 (which is taken up below). 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 = numYmodes is a convenience that allows a single variable to unambiguously hold an inter or intra prediction mode.

```
typedef enum
{
   MV_nearest = numYmodes, /* 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 */

   numMVrefs = MV_split + 1 - MV_nearest
}
MVref;

const TreeIndex MVrefTree [2 * (numMVrefs - 1)] =
{
 -MV_zero, 2,           /* zero = "0" */
  -MV_nearest, 4,       /* nearest = "10" */
   -MV_near, 6,            /* near = "110" */
     -MV_new, -MV_split    /* new = "1110," split = "1111" */
};

/* Internal representation of a motion vector */

typedef struct { int16 row, col;}  MV;
```

16.3 Mode and Motion Vector Contexts

The probability table used to decode the MVref, along with three reference motion vectors used by the selected mode, are calculated, using already-decoded motion vectors in (up to) 12 nearby blocks, by a fairly elaborate process best described by the reference implementation itself (the function "FindNearMVs" in the file "findnearmv.c").

We do make one remark regarding this function, though. Unlike some other context prediction methods used by the reference decoder, FindNearMVs explicitly excludes "invisible" blocks implicitly referenced by macroblocks lying on the top or left edges of the frame, not caring about any synthetic records of non-existent motion vectors that might be maintained (however, such records, having zero vectors, could reasonably be used for the subblock motion vector contexts discussed toward the end of this chapter). Furthermore, blocks lying within the first visible macroblock (at the upper left corner of the frame) are also excluded from consideration.

FindNearMVs actually calculates four things:

```
   MV nearest, near; /* motion vectors to be used by modes of the same name
*/
   MV best;      /* base of explictly-coded motion vectors */

   int ct[4]; /* weighted census of zero, nearest, near, and other vectors */
```

The MVref probability table is derived from the census as follows:

```
const Prob ModeContexts [31] [4] =
{
  {   3,    3,    1,  246},
  {   7,   89,   66,  239},
  {  10,   90,   78,  238},
  {  14,  118,   95,  241},
  {  14,  123,  106,  238},
```

```
  {  20,  140,  109,  240},
  {  13,  155,  103,  238},
  {  21,  158,   99,  240},
  {  27,   82,  108,  232},
  {  19,   99,  123,  217},
  {  45,  139,  148,  236},
  {  50,  117,  144,  235},
  {  57,  128,  164,  238},
  {  69,  139,  171,  239},
  {  74,  154,  179,  238},
  { 112,  165,  186,  242},
  {  98,  143,  185,  245},
  { 105,  153,  190,  250},
  { 124,  167,  192,  245},
  { 131,  186,  203,  246},
  {  59,  184,  222,  224},
  { 148,  215,  214,  213},
  { 137,  211,  210,  219},
  { 190,  227,  128,  228},
  { 183,  228,  128,  228},
  { 194,  234,  128,  228},
  { 202,  236,  128,  228},
  { 205,  240,  128,  228},
  { 205,  244,  128,  228},
  { 225,  246,  128,  228},
  { 233,  251,  128,  228}
};
   Prob MVrefP [numMVref - 1];

   MVrefP [0] = ModeContexts [ct[0]] [0];
   MVrefP [1] = ModeContexts [ct[1]] [1];
   MVrefP [2] = ModeContexts [ct[2]] [2];
   MVrefP [3] = ModeContexts [ct[2]] [3];
```

Note that ct[3], the count of "other" vectors, is not used. As is implied by the dimensions of the ModeContexts array, the maximum possible "weighted census" value is 30. Once MVrefP is established, the MVref is decoded as usual:

```
   mvr = (MVref) TreedRead( d, MVrefTree, MVrefP);
```

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

```
   MV_nearest     use the "nearest" vector returned by FindNearMVs
   MV_near        use the "near"    vector returned by FindNearMVs
   MV_zero        use a zero vector
```

MV_zero simply predicts the current macroblock from the corresponding macroblock in the prediction frame. The next mode (MV_new) is followed by an explicitly-coded motion vector (the format of which is described in the next chapter) that is added (componentwise) to the "best" reference vector returned by FindNearMVs and applied to all 16 subblocks.

16.4 Split Prediction

The remaining mode (MV_split) causes multiple vectors to be applied to the Y subblocks. It is immediately followed by a "partition" specification that determines how many vectors will be specified

and how they will be assigned to the subblocks. The possible partitions, with indicated subdivisions and coding tree, are as follows.

```
typedef enum
{
   MV_TopBottom,  /* two pieces {0...7} and {8...15} */
   MV_LeftRight,  /* {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_numPartitions
}
MVpartition;

const TreeIndex MVpartitionTree [2 * (MVnumPartition - 1)] =
{
 -MV_16, 2,                    /* MV_16 = "0" */
  -MV_Quarters, 4,                /* MV_Quarters = "10" */
   -MV_TopBottom, -MV_LeftRight     /* TopBottom = "110," LeftRight = "111"
*/
};
```

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

```
   const Prob MVpartitionProbs [MVnumPartition - 1] = { 110, 111, 150};

   part = (MVpartition) TreedRead( d, MVpartitionTree, MVpartitionProbs);
```

After the partition come two (for MV_TopBottom or MV_LeftRight), four (for MV_Quarters), or sixteen (for MV_16) subblock inter-prediction modes. These modes occur in the order indicated by the comments above and are coded as follows. As was done for the macroblock-level modes, we offset the mode enumeration so that a single variable may unambiguously hold either an intra or inter subblock mode.

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

   numSubMVref
};
subMVref;

const TreeIndex subMVrefTree [2 * (numSubMVref - 1)] =
{
 -LEFT4X4, 2,         /* LEFT = "0" */
  -ABOVE4X4, 4,          /* ABOVE = "10" */
   -ZERO4X4, -NEW4X4 /* ZERO = "110," NEW = "111" */
};

/* Constant probabilities and decoding procedure. */

const Prob subMVrefProb [numSubMVref - 1] = { 180, 162, 25};
```

```
subRef = (subMVref) TreedRead( d, subMVrefTree, subMVrefProb);
```

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. Finally, 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 "MV_new," 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 "FindNearMVs" 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 VP7 datastream: Applied to whole macroblocks in "MV_new" 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 nonzero, is followed by a boolean sign. V may take any value between -255 and +255 inclusive.

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

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

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

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

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

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


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

const TreeIndex SmallMVtree [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 readMVcomponent( BoolDecoder *d, const MV_CONTEXT *mvc)
{
   const Prob * const p = (const Prob *) mvc;
   int A = 0;

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

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

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

      i = 7;
      do { A += readBool( 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 & 240)  ||  readBool( d, p [MVPbits + 3]))
         A += 8;
   }
   else                              /* 0 <= A <= 7 */
      A = TreedRead( d, SmallMVtree, p + MVPshort);

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

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? P(7), that is, a Bool followed by a 7-bit probability specification if true.

As with other dynamic probabilities used by VP7, the updates remain in effect until the next key frame or until replaced via another update.

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

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

const MV_CONTEXT MVupdateProbs [2] =
{
   {                                    /* row update probs */
      237,
      246,
      253, 253, 254, 254, 254, 254, 254,
      254, 254, 254, 254, 254, 250, 250, 252
   },
   {                                    /* column update probs */
      231,
      243,
      245, 253, 254, 254, 254, 254, 254,
      254, 254, 254, 254, 254, 251, 251, 254
   }
};

/* Default MV decoding probabilities. */

const MV_CONTEXT DefaultMVcontext [2] =
{
   {                                    /* row */
      162,                              /* is short */
      128,                              /* sign */
       225, 146, 172, 147, 214,  39, 156,       /* short tree */
       247, 210, 135,  68, 138, 220, 239, 246      /* long bits */
   },
   {                                    /* same for column */
      164,
      128,
      204, 170, 119, 235, 140, 230, 228,
      244, 184, 201,  44, 173, 221, 239, 253
   }
};

/* 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 readMV( BoolDecoder *d)
{
   MV v;
   v.row = (int16) readMVcomponent( d, mvc);
   v.col = (int16) readMVcomponent( 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 updateMVcontexts( boolDecoder *d)
{
   int i = 0;
   do {                          /* component = row, then column */

      const Prob *up = MvUpdateProbs[i];  /* update probs for component */
      Prob *p = mvc[i];              /* start decode tbl "" */
      Prob * const pstop = p + MVPcount;  /* end decode tbl "" */
      do {
         if( readBool( d, *up++))      /* update this position */
         {
            const Prob x = readLiteral( d, 7);

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

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

18. INTER-PREDICTION BUFFER CALCULATION

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 and loop filtering.

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

18.1 Bounds On, and Adjustment of, Motion Vectors

Although, through repeated addition of references, it is physically possible within the VP7 format to encode arbitrarily large motion vectors, it is part of the definition of the format that, after addition of the reference vector to the explicitly-coded offset as described in the previous two chapters, the absolute value of the resulting components never exceeds 255, that is, 63 3/4 luma pixels or 31 7/8 chroma pixels.

Consequently, the "halo" dimensions discussed in Chapter 5 can now be established. Inter-prediction will be simplest if a "full-size halo" of 67 luma pixels is used (allowing for a 64-pixel motion vector plus three extra prediction pixels for the interpolation detailed below). Alternatively, a smaller halo large enough to at least accommodate prediction blocks straddling the "visible" and "pad" pixel could be used, together with explicit handling of the exceptional case of prediction blocks completely outside the visible frame.

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

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

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

```
int avg( int c1, int c2, int c3, int c4) {

   int s = c1 + c2 + c3 + c4;

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

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

The chroma vectors for corresponding U and V subblocks (that is, U and V subblocks occupying the same area within the image) are of course identical.

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 subpixel interpolation processes for the subblock.

It is here (and only here) that any prediction-pitch override must be respected, that is, the position of the subblock relative to the calculated origin and the “stride” (or separation between what are to be considered vertically-adjacent pixels) must be taken in accordance with which of the 8 prediction pitches is in effect for this macroblock. The strides associated to the different pitches, together with the relative positioning of subblocks, are described in detail in Chapter 10 above.

18.3 Subpixel Interpolation

As was also discussed in Chapter 10, the subpixel 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, which 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." For example, if the pitch is "X2," we may predict an "even" subblock starting from either an "even" or "odd" line. This is intentional and desirable.

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" associated with the pitch). 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 associated to the inter-prediction pitch, as are all vertical displacements used here.

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

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

The exact implementation of subsampling is as follows.

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

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

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

uint8 interp(
   const int fil[6], /* filter to apply */
   const uint8 *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(
   uint8 temp[9][4], /* 9 rows of 4 (intermediate) destination values */
   const uint8 *p,      /* subblock origin in prediction frame */
   int s,            /* vertical stride to be used in prediction frame */
   uint hfrac        /* 0 <= horizontal displacement <= 7 */
) {
   const int * const fil = filters [hfrac];

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

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


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

void Vinterp(
   uint8 final[4][4],      /* 4 rows of 4 (final) destination values */
   const uint8 temp[9][4],
   uint vfrac           /* 0 <= vertical displacement <= 7 */
) {
   const int * const fil = filters [vfrac];

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

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

18.4 Filter Properties

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

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

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

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

It is true that the theoretically correct subsampling procedure, as well as any approximation thereof used in the real world, is always given by a "translation-invariant" weighted sum (or filter) similar to that used by VP7. It is also true that the reconstruction error made by such a filter can be simply represented as a "multiplier" in the freqency 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") and 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, more generally, 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 VP7 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 freqency response exceeds one).

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

To get slightly more specific, the filters chosen by VP7 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.

TL
A[0]
A[1]

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 prception 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. GOLDEN FRAME UPDATE

The update of golden frames applies after loop-filtering is complete.

A reconstructed key frame is always copied into the golden frame buffer.

If indicated in its frame header, the entirety of a reconstructed interframe is copied into the golden frame buffer. Otherwise, the portion of the reconstructed interframe buffer corresponding to each macroblock is copied to the golden frame buffer if indicated by the macroblock's partial golden frame update bool (one of the macroblock-level "features" discussed above).

Finally, edge pixels should be propagated to the "halos" of both the golden and current frame as discussed above and the decoder should swap its references to the current and last frames in preparation for decoding the next frame.

DOCUMENT REVISION HISTORY

Document Version	Description	Name/Date
1.0	Created the document.	Tim Murphy 3/21/2005
1.1	Reformatted for On2 standard template.	John Luther 3/21/2005
1.2	Added Macroblock Features, DCT Coefficient Decoding, Golden Frame Update sections	Tim Murphy 3/23/2005
1.3	Added Interframe Macroblock Prediction Records, Motion Vector Decoding sections.	Tim Murphy 3/24/2005
1.4	Added Inverse DCT Calculation and Macroblock Reconstruction and Inter-prediction Buffer Calculation sections.	Tim Murphy 3/27/2005
1.5	Added subheadings, final formatting.	Tim Murphy, John Luther 3/28/2005