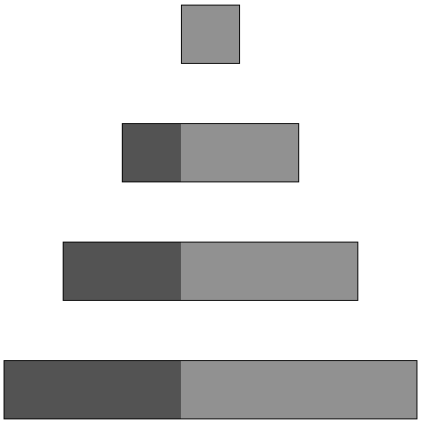
DIGITAL SUBBAND VIDEO

VERSION 2.0

REVISION 1

October, 2024

2024 EMMIR
ENVEL GRAPHICS



Contact Information:

GitHub : https://github.com/LMP88959
YouTube: https://www.youtube.com/@EMMIR_KC/videos
Discord: https://discord.com/invite/hdYctSmyQJ

LICENSING

Specification License

Software License

The accompanying software was designed and written by EMMIR, 2024 of Envel Graphics.
If you release anything with it, a comment in your code/README saying where you got this code would be a nice gesture but it’s not mandatory.

Please note this document is not intended to be formal or comprehensive. It would be virtually impossible to create a fully functional DSV2 decoder explicitly using this document as reference.

CONVENTIONS

Pseudo code is often written in the C or a pseudo-C language.
All data types are assumed to be 32-bit integers unless otherwise specified.

A prefix of 0x for a number means that the number is in hexadecimal format. For example, 0x55 is 85 in decimal.

DEFINITIONS

Block: a rectangular subregion of a plane of a frame.

Chroma/UV: the two planes of a frame that represent the color of the image at every point.

Frame: three two-dimensional grids of pixels representing the video at a certain point in time.

Half-Pixel Filter: a filter used to generate a new image as though the original image had been shifted over by half a pixel.

In-Loop Filter: Refers to using the filtered frame as the reference for the next frame rather than a post-processing filter.

Interleaved Exponential-Golomb Code: A form of variable length coding which can be more efficiently decoded due to its structure.

Inter Frame: a frame of video that relies on another frame to be correctly reconstructed.

Intra Frame: a frame of video that does not rely on another frame to be correctly reconstructed.

Luma/Y: the plane of a frame that represents the brightness of the image at every point within it.

Motion Compensation: applying temporal prediction in order to reconstruct an inter frame.

Plane: a single component of the video, also known as a channel. Luma is its own plane, chroma consists of two planes: U, and V.

Variable Length Code: A way to represent an arbitrarily large integer in a variable number of bits.

TABLE OF CONTENTS

A. INTRODUCTION

A.1 Description

Digital Subband Video 2 (DSV2) is a digital video compression specification designed for efficient software decoding and playback on consumer hardware while offering compression performance comparable to Moving Picture Experts Group's MPEG-4 Part 2 and Part 10 video standards. DSV2 was designed for medium-low to medium-high bit rates.

DSV2 is highly portable and does **not** require use of:

- data types with widths greater than 32 bits
- floating point data types or literals
- a specific operating system, processor
- any 3rd party libraries

A.2 Brief Technology Overview

DSV2 is a lossy hybrid video codec[1] which utilizes multiple subband filters and block motion compensation to achieve spatial and temporal compression.

Overview:

- operates on YCC video (luma with two chroma components)
  - supports chroma subsampling of 4:4:4, 4:2:2, 4:2:0, and 4:1:1
  - only 8 bits of depth per component supported
  - no explicit support for interlaced video
- frame sizes must be divisible by two
- subband transform
  - two-dimensional Haar + smoothing filter
  - various hybrids of 3 and/or 5 tap filters with varying coefficients
  - full decomposition (recurse on LL band until 1 pixel remaining)
- block motion compensation
  - half and quarter pixel precision
  - block size can vary but all blocks in one frame must be the same size
  - intra blocks can be split into 4 sub-blocks
  - in-loop filtering
- adaptive quantization
- hierarchical zero coefficient coding (HZCC) using run-length encoding and interleaved exponential-golomb codes
- intra (I) frames and inter (P) frames with variable length closed GOP (group of pictures)

[1] Hybrid refers to the use of image transform coding and motion estimation/compensation. This is in contrast to coding a video as if it were a 3D signal.

A.3 Improvements and New Features

DSV2's fundamentals and architecture are very similar to DSV1. The substantial video coding improvements come mostly from a few key new features.

New Features:

- in-loop filtering after motion compensation
- more adaptive quantization potential
- skip blocks for temporal stability
- better motion compensation through Expanded Prediction Range Mode (EPRM)
- new subband filters + support for adaptive subband filtering
- quarter pixel compensation
- psychovisual optimizations in the codec and encoder design

B. BITSTREAM DECODING

The fundamental unit of digital information is a binary digit, or bit, which can be either a 0 or 1. A byte is defined as 8 bits.

DSV operates on bits exclusively and only uses these types of binary encoding:

Encoding type	Description
Raw	The bits stored directly represent the binary digits of a number.
Interleaved Exponential-Golomb Code (UEG)	A method of encoding arbitrarily large positive (including zero) integers in a variable number of bits.
Signed Interleaved Exponential-Golomb Code (SEG)	A method of encoding arbitrarily large integers in a variable number of bits.
Non-zero Interleaved Exponential-Golomb Code (NEG)	A method of encoding arbitrarily large non-zero integers in a variable number of bits.

Some parts of the DSV bitstream contain a form of data encoding called Zero Bit Run-Length-Encoding (ZBRLE). ZBRLE operates on bits and consists of a concatenated sequence of UEG codes. The value of each decoded UEG code is the number of zero bits that it represents. If no zero bits remain, the decoded bit is a one. Decoding a ZBRLE bit is described in the following pseudo code:

```
if (number_of_zeros_remaining[2] == 0) {
    number_of_zeros_remaining = read_UEG();
    return (number_of_zeros_remaining == 0 ? 1 : 0);
}
number_of_zeros_remaining--;
return (number_of_zeros_remaining == 0 ? 1 : 0);
```

A DSV stream consists of a number of packets. Packets contain a header and a payload. The data in the following tables in each subsection is sequential in the bitstream.

[2] *number_of_zeros_remaining* is a state variable associated with the ZBRLE stream being decoded.

B.1 Packet Header

Data	Contains	Description
8 bits	ASCII character ‘D’ Hex 0x44	First character of the Four Character Code (FourCC)
8 bits	ASCII character ’S’ Hex 0x53	A method of encoding arbitrarily large positive (including zero) integers in a variable number of bits.
8 bits	ASCII character ‘V’ Hex 0x56	A method of encoding arbitrarily large integers in a variable number of bits.
8 bits	ASCII character ‘2’ Hex 0x32	A method of encoding arbitrarily large non-zero integers in a variable number of bits.
8 bits	Minor version number	Minor version number of the DSV specification the encoder adhered to.
8 bits	Packet type	A bit pattern which describes the contents packet.
32 bits	Previous link offset	Length (in bytes) of the previous packet.
32 bits	Next link offset	Length (in bytes) of the current packet.

B.1.1 Packet Type

Packet Type	Description
Equal to 0x00	Packet is metadata.
Equal to 0x10	Packet is an end of stream marker.
Packet Type bit 0x04 is set	Packet is a picture. If Packet Type bit 0x01 is set, it has a reference picture. If Packet Type bit 0x02 is set, it is a reference picture.

B.2 Packet Payload

B.2.1 Metadata Packet

Data	Contains	Description
UEG encoded	Width of video frame	Horizontal resolution in pixels.
UEG encoded	Height of video frame	Vertical resolution in pixels.
UEG encoded	Chroma subsampling	4:4:4 = 0x00 4:2:2 = 0x04 4:2:0 = 0x05 4:1:1 = 0x08
UEG encoded	Frames per second (FPS) numerator	Used in combination with the FPS denominator to define the (potentially fractional) frame rate of the video. FPS = numerator / denominator
UEG encoded	Frames per second (FPS) denominator	Used in combination with the FPS numerator to define the (potentially fractional) frame rate of the video. FPS = numerator / denominator
UEG encoded	Aspect ratio numerator	Used in combination with the aspect ratio denominator to define the aspect ratio of the video. Aspect ratio = numerator / denominator
UEG encoded	Aspect ratio denominator	Used in combination with the aspect ratio numerator to define the aspect ratio of the video. Aspect ratio = numerator / denominator

B.2.2 End of Stream Packet

Data	Contains	Description
None	Nothing	End of stream packet has no payload.

B.2.3 Picture Packet

Data	Contains	Valid Range	Description
32 bits	Frame number	All positive integers, including zero.	ID of the frame. Ideally sequential/ chronological.
UEG encoded	log2(block_width) - 4	[0, 1]	Motion block width can be calculated by multiplying 16 by two to the power of the decoded number. In C: **`block_width = 16 << decoded`**
UEG encoded	log2(block_height) - 4	[0, 1]	Motion block height can be calculated by multiplying 16 by two to the power of the decoded number. In C: **`block_height = 16 << decoded`**
1 bit	Stable statistic	[0, 1]	1 if zeros are used as the end-of-run symbol 0 if ones are used as the end-of-run symbol
1 bit	**If intra frame,** maintain statistic. **If inter frame**, mode statistic.	[0, 1]	1 if zeros are used as the end-of-run symbol 0 if ones are used as the end-of-run symbol
1 bit	**If intra frame,** ringing statistic. **If inter frame**, EPRM statistic.	[0, 1]	1 if zeros are used as the end-of-run symbol 0 if ones are used as the end-of-run symbol
12 bits	Quantization parameter ***Q***	[0, 4095]	Frame quantization parameter.

The number of motion blocks in the horizontal and vertical direction can be derived like so:

```
horiz_blocks = (frame_width + block_width - 1) / block_width
vert_blocks = (frame_height + block_height - 1) / block_height
```

Where *frame_width* and *frame_height* come from the metadata packet.

B.2.3.1 Stability Blocks

Every picture has a stability block subsection, in the case of P frames, it represents skipped blocks. Note that the ZBRLE bits should be modified appropriately based off of the corresponding statistic read in the **B.2.3 Picture Packet** table.

Data	Contains	Valid Range	Description
UEG encoded Align to next byte afterwards.	Length of this subsection in bytes.	All positive integers, including zero.	Number of bytes to skip to reach the next subsection of the picture packet.
ZBRLE encoded	One bit for each motion block in the frame. In I frames: used for adaptive quantization. In P frames: used to flag a block as a skip block.	N/A	Compute (horiz_blocks * vert_blocks) to determine how many ZBRLE bits must be read.

B.2.3.2 Ringing Blocks

Only I frames have a ringing blocks subsection. Note that the ZBRLE bits should be modified appropriately based off of the corresponding statistic read in the **B.2.3 Picture Packet** table.

Data	Contains	Valid Range	Description
UEG encoded Align to next byte afterwards.	Length of this subsection in bytes.	All positive integers, including zero.	Number of bytes to skip to reach the next subsection of the picture packet.
ZBRLE encoded	One bit for each motion block in the frame.	N/A	Compute (horiz_blocks * vert_blocks) to determine how many ZBRLE bits must be read.

B.2.3.3 Maintain Blocks

Only I frames have a maintain blocks subsection. Note that the ZBRLE bits should be modified appropriately based off of the corresponding statistic read in the **B.2.3 Picture Packet** table.

Data	Contains	Valid Range	Description
UEG encoded Align to next byte afterwards.	Length of this subsection in bytes.	All positive integers, including zero.	Number of bytes to skip to reach the next subsection of the picture packet.
ZBRLE encoded	One bit for each motion block in the frame.	N/A	Compute (horiz_blocks * vert_blocks) to determine how many ZBRLE bits must be read.

B.2.3.4 Motion Data

Only exists for pictures that have a reference (inter frames). Align to next byte before starting the decoding process. The subsections occur in order, data for each motion block is not interleaved. First, a UEG encoded integer is read before each subsection. This integer is the length in bytes of the subsection that follows it. The stream is aligned to the next byte after each UEG decoded. Note that the ZBRLE bits should be modified appropriately based off of the corresponding statistic read in the **B.2.3 Picture Packet** table.

Subsection	Data	Contains	Description
Mode	ZBRLE encoded	Block inter/ intra modes.	0x00 = block is INTER 0x01 = block is INTRA
EPRM	ZBRLE encoded	Expanded Prediction Range Mode flags.	0x00 = block is in limited prediction range mode 0x01 = block is in expanded prediction range mode
Motion vector X components	SEG encoded	X component of each block's motion vector.	Exists only if block is INTER. If SKIP then set component to zero and do not read any data. If NOT SKIP, read the data and reconstruct the actual component by adding its prediction to the decoded value.
Motion vector Y components	SEG encoded	Y component of each block's motion vector.	Exists only if block is INTER. If SKIP then set component to zero and do not read any data. If NOT SKIP, read the data and reconstruct the actual component by adding its prediction to the decoded value.
Intra sub-block masks	Raw - Described below	Bit masks of the sub-blocks.	Exists only if block is INTRA.

Motion Vector Prediction

Motion vector components for a given block are predicted from the vector components of its left, upper, and upper left neighbors in the block grid. If the current block is on the left edge, top edge, or top left, the unavailable neighbors' component used for prediction will be zero.

The prediction using the three neighbors is obtained by:

```
motion_vector_prediction(LEFT, TOP, TOP_LEFT)
{
    dif = LEFT + TOP - TOP_LEFT;
    lft = absolute_value(dif - LEFT);
    top = absolute_value(dif - TOP);
    if (lft < top) {
        return LEFT;
    }
    return TOP;
}
```

Intra Sub-Block Mask Decoding

Intra blocks are split into four sub-blocks. Each sub-block can then either be coded as intra or inter with a zero motion vector.

1	2
4	8

Mask value of each sub-block

Decoding the mask is done by first reading a bit. If the bit was a 1 then all the sub-blocks are marked as intra, otherwise 4 bits are read and assigned to the mask. This is because of the observation that intra blocks where all sub-blocks are intra as well are much more common. Pseudo code for the described behavior:

```
if (read_bit() == 1) {
    submask = ALL_INTRA;
} else {
    submask = read_4_bits();
}
```

B.2.3.5 Image Data

Every picture has an image data subsection.

Plane Decoding

Data	Contains	Description
32 bits	Length in bytes of encoded plane.	Amount of bytes to skip to get to the next color plane.
Complex - Coefficient Decoding	Encoded coefficients of plane.	Subband coefficients of the color plane.

The above is performed for each of the three color frames, making sure to byte align the stream after each plane.

Coefficient Decoding

Data	Contains	Description
SEG encoded	LL coefficient of the entire frame.	All the low pass energy of the image. It is not quantized and often has a very large magnitude.
Complex - HZCC	Encoded coefficients of plane.	Subband coefficients of the color plane.
8 bits	End of Plane Symbol Hex 0x55	Check decoded value matches 0x55 after HZCC decoding to see if there was an error in the plane's bit stream.

Decoding Hierarchical Zero Coefficient Coding (HZCC) Data

Align the stream to the next byte before beginning.

Data	Contains	Description
32 bits	Number of runs.	Total number of zero runs in the coefficients.
UEG encoded	Length of first zero run.	How many of the first coefficients are zero.

Begin loop through the ‘LL’ subband. If there are no more zeros left in the current run, read the following data to get the next run and then continue the loop.

Data	Contains	Description
UEG encoded	Length of next zero run.	How many of the next set of coefficients are zero.
NEG encoded	Non-zero coefficient.	Quantized, needs to be dequantized.

After decoding the ‘LL’ band, begin decoding the 3 highest frequency subbands in order from lowest to highest. The decoding process is identical to that used in the ‘LL’ subband, described in the table above.

Once all subbands are decoded, align to the next byte and continue with the error check in the **Coefficient Decoding** table. Set the first coefficient of the plane to be the LL coefficient decoded in the **Coefficient Decoding** table.

Repeat for each plane until image data is fully decoded.

NOTE: **all** subband coefficients must be stored as signed 32-bit integers.

At this point, no more data needs to be read from the stream for this frame.

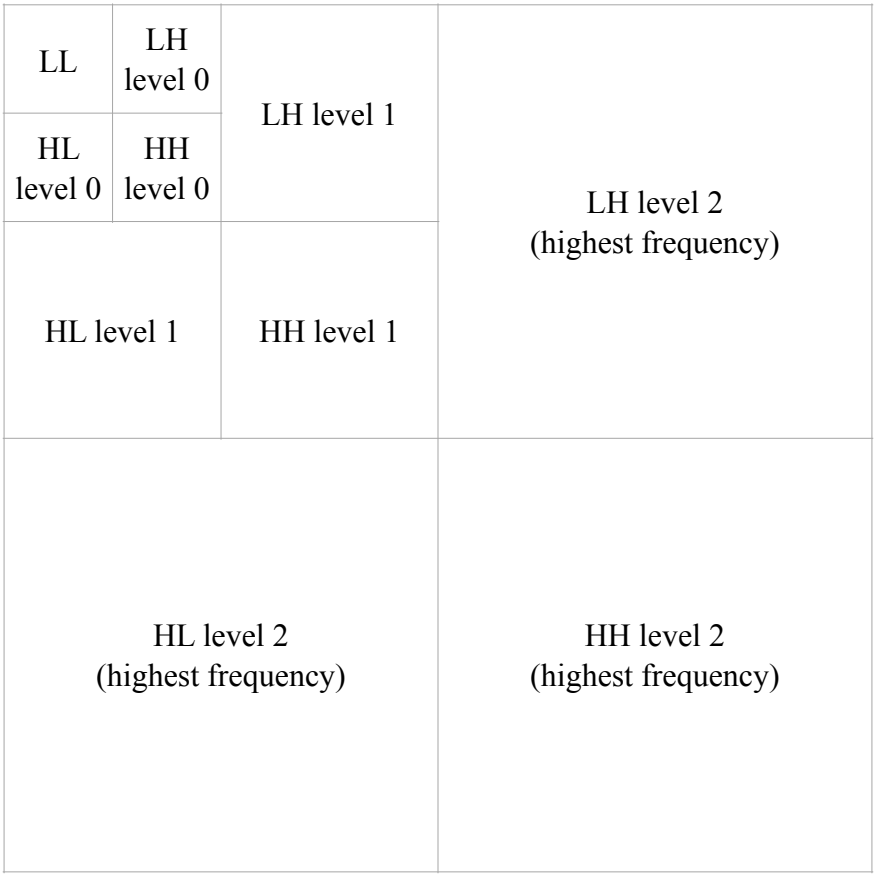
C. IMAGE RECONSTRUCTION

C.1 Subband Order and Traversal

Only the three highest frequency subbands are treated specially in DSV. The other lower frequency subbands get grouped together into the LL region in the context of HZCC and quantization. The subbands are traversed in raster scan order. Raster scan order is left to right, top to bottom. Pseudo code for such a traversal is given below:

```
for (y = top; y < bottom; y++) {
    for (x = left; x < right; x++) {
        /* data is at [x, y] */
    }
}
```

X increases horizontally, Y increases vertically



DSV subband structure in 2D

NOTE: when computing subband dimensions through subdivision, make sure to round up to the next integer.

C.2 Dequantization

In DSV, each level of the structure is quantized in a different fashion.
The quantization parameter ***Q*** is passed into the HZCC decoder and limited to **512** for chroma planes. The minimum quantization parameter ***MINQUANT*** is **16** for all levels except the highest frequency level.

Some important data needed for correct dequantization:

Data	Denoted As	Contains
Stability Blocks	boolean = STABLE(x, y)	Bit array with enough elements for every block. A value of 0 at [x, y] means that block was determined to be unstable by the encoder. A value of 1 at [x, y] means that block was determined to be stable by the encoder.
Maintain Blocks	boolean = MAINT(x, y)	Bit array with enough elements for every block. A value of 0 at [x, y] means that block should not be maintained. A value of 1 at [x, y] means that block should be maintained.
Skip Blocks	boolean = SKIP(x, y)	Bit array with enough elements for every block. A value of 0 at [x, y] means that block was not determined to be skipped by the encoder. A value of 1 at [x, y] means that block was determined to be skipped by the encoder.
Intra Blocks	boolean = INTRA(x, y)	Bit array with enough elements for every block. A value of 0 at [x, y] means that block was marked as inter. A value of 1 at [x, y] means that block was marked as intra.
Is Inter Frame	boolean = IS_INTER	Boolean value denoting whether this frame was an intra frame or an inter frame. false = intra frame true = inter frame

NOTE the fixed point precision for determining what block a pixel (x, y) lies in is **14**.

C.2.1 Dequantization Functions

Dequantization is performed through dead-zone dequantization for all subbands besides the highest frequency subband. The pseudo code for the dequantization functions is given below:

```
dequantize_lower_frequency(VALUE, Q)
{
    if (VALUE < 0) {
        return (VALUE * Q) - (Q / 2);
    }
    return (VALUE * Q) + (Q / 2);
}

dequantize_highest_frequency(VALUE, Q)
{
    if (VALUE < 0) {
        return -((-VALUE * Q) + ((2 ^^ Q) / 2));
    }
    return (VALUE * (2 ^^ Q) + ((2 ^^ Q) / 2);
}
```

C.2.2 Quantization Parameter Derivation

The *dynamic_adjust* function is designed to 'push back' and increase ***Q*** when its value is large.

```
dynamic_adjust(Q, F)
{
   return Q + ((4095 - q) / (2 ^^ (12 - F)));
}
```

Q is updated (and applies as the base ***Q*** for the whole image) after the chroma limiting as follows:

```
if (IS_INTRA) Q = dynamic_adjust(Q, 6);
else          Q = dynamic_adjust(Q, 5);
```

The modified quantization parameter ***MQ*** for a given level is derived as follows.

If level is not highest frequency level and is not 'LL':

```
/* level is based on C.1's diagram */
get_quant_lower_frequency(Q, level, subband)
{
   MQ = Q;
   if (IS_INTER) {
      if (LUMA) return dynamic_adjust(MQ, 8) / 8;
      return dynamic_adjust(MQ, 4) / 4;
   }
   MQ = dynamic_adjust(MQ, 8);
   if (subband != HH) {
      MQ /= 2;
   }
   if (MQ < MINQUANT) {
      return MINQUANT;
   }
   return MQ;
}
```

If level is highest frequency level:

```
get_quant_highest_frequency(MQ)
{
   if (LUMA) {
      if (IS_INTER) {
         MQ = MAX(MQ - 3, 4);
         MQH = MQ
      } else {
         MQ = MAX(MQ, 4)
         MQH = MAX(MQ - 4, 4);
      }
   } else {
      MQ = MAX(MQ, 4);
      MQH = MQ;
   }
   return (MQ, MQH);
}
```

The ***CLAMP*** function is defined as:

```
CLAMP(x, min, max) /* clamp x between [min, max] */
     return ((x) < (min) ? (min) : ((x) > (max) ? (max) : (x)))
```

C.2.3 LL Subband Dequantization

Each NEG encoded coefficient that is decoded in the LL subband is dequantized as follows:

```
for (y = top; y < bottom; y++) {
   for (x = left; x < right; x++) {
      MQ = get_quant_lower_frequency(Q, LL);
      V = read_NEG();
      decoded[x,y] = dequantize_lower_frequency(V, MQ);
   }
}
```

C.2.4 Higher Level Subband Dequantization

The higher levels take the 2D position of the coefficient into account for deriving the true ***MQ***, denoted as ***TMQ*** from now on.

```
TMQ_for_position(X, Y, MQ)
{
   if (IS_INTER) {
       if (SKIP(X, Y)) return MQ * 2;
       if (INTRA(X, Y) && LUMA) return MQ;
       return MQ + (MQ / 2);
   }
   if (CHROMA && level == 1 && (STABLE(X, Y) || MAINT(X, Y)) {
       MQ *= 2;
   }
   switch (MAINT | STABLE) {
      case STABLE(X, Y):
         if (level == 0) break;
         MQ /= 2; break;
      case MAINT(X, Y):
         if (level == 0) break;
         MQ /= 3; break;
      case STABLE(X, Y) && MAINT(X, Y):
         if (level == 0) MQ -= MQ / 4; break;
         MQ /= 4; break;
   }
   return MQ + 1;
}
```

Each NEG encoded coefficient that is decoded in the higher level subbands is dequantized as follows:

NOTE: this does not include the highest level.

```
highest_level = 3;

for (level = 0; level < (highest_level - 1); level++) {
for (each subband in order LH, HL, HH) /* NOTE no open brace */
   for (y = top; y < bottom; y++) {
      for (x = left; x < right; x++) {
         MQ = get_quant_lower_frequency(Q, level, subband);
         TMQ = TMQ_for_position(x, y, MQ);
         V = read_NEG();
         decoded[x,y] = dequantize_lower_frequency(V, TMQ);
      }
   }
}
```

C.2.5 Highest Level Subband Dequantization

The highest level also takes the 2D position of the coefficient into account for deriving the ***TMQ***.

```
MQ = integer_log_base_2(Q);
MQ += (12 - MQ) / 4;
level = highest_level - 1;
for (each subband in order LH, HL, HH) {
   for (y = top; y < bottom; y++) {
      for (x = left; x < right; x++) {
         (MQ, MQH) = get_quant_highest_frequency(MQ);
         if (STABLE(x, y)) {
            TMQ = MQH;
         } else {
            TMQ = MQ;
         }
         if (IS_INTRA && subband == HH) TMQ += 2;
         V = read_NEG();
         decoded[x,y] = dequantize_highest_frequency(V, TMQ);
      }
   }
}
```

C.3 Subband Transforms

DSV performs a full decomposition. A full decomposition is a recursive subband transform on the LL subband of the structure until the LL subband is a single pixel.

DSV uses a few different transforms, the *Haar Transform* (HT) and 3 or 5 tap filters.

The ***MIN*** function is defined as:

```
MIN(a, b) ((a) < (b) ? (a) : (b))
```

The ***MAX*** function is defined as:

```
MAX(a, b) ((a) > (b) ? (a) : (b))
```

The functions ***ROUND2*** and ***ROUND4*** simply divide the argument by the number in the function's name while also rounding away from zero (properly accounting for negatives).
For example:

```
ROUND4(V)
    if (V < 0) {
        return -(((-V) + 2) / 4);
    }
    return (V + 2) / 4;
```

C.3.1 Haar Transform

C.3.1.1 Simple Inverse Transform

The HT is the simplest transform, it operates on 2x2 blocks of pixels.

Given a 2x2 block of pixels in the form:
{ x0, x1,
  x2, x3 }
The simple inverse HT in DSV is defined as follows:

```
x0 = (LL + LH + HL + HH) / 4;
x1 = (LL - LH + HL - HH) / 4;
x2 = (LL + LH - HL - HH) / 4;
x3 = (LL - LH - HL + HH) / 4;
```

C.3.1.2 Filtered Inverse Transform

The filtered inverse HT in DSV is defined as follows:

```
if (NOT THE FIRST OR LAST COEFFICIENT IN ROW) {
    lp = p_LL; /* previous LL on X axis (x - 1) */
    ln = n_LL; /* next LL on X axis (x + 1) */
    mx = LL - ln; /* find difference between LL values */
    mn = lp - LL;
    if (mn > mx) SWAP(mn, mx)
    mx = MIN(mx, 0); /* must be negative or zero */
    mn = MAX(mn, 0); /* must be positive or zero */
    /* if they are not zero, then there is a potential
     * consistent smooth gradient between them */
    if (mx != mn) {
        n = ROUND2(CLAMP(ROUND4(lp - ln), mx, mn) - (LH * 2));
        LH += CLAMP(n, -HQP, HQP); /* nudge LH to smooth it */
    }
}
/* do the same as above but in the Y direction, but nudging HL
 * instead of LH */
x0 = (LL + LH + HL + HH) / 4;
x1 = (LL - LH + HL - HH) / 4;
x2 = (LL + LH - HL - HH) / 4;
x3 = (LL - LH - HL + HH) / 4;
```

The ***HQP*** variable is approximately half the quantization value used for the subband. Due to the adaptive quantization, we cater to the regions that are stable and smooth those properly since they are more likely to be noticed when improperly filtered.

HQP for given ***Q*** can be derived as follows:

```
get_HQP(Q)
{
    if (LUMA) {
        return Q / 8;
    }
    return Q / 2;
}
```

NOTE: the Haar subband transforms must properly account for odd dimensions by treating any pixel or coefficient in the 2x2 block that lies outside of the image as though it has a value of zero.

C.3.2 The 3 and 5-Tap Filters

These longer transforms are ONLY done to levels 1-4 and ONLY done to INTRA frames.
For the sake of simplicity, only the inverse transforms are defined here. The forward transforms are essentially the same but done in reverse order and performing the opposite arithmetic operation for each sample (i.e highpass -> lowpass -> scale+pack instead of unscale+unpack -> lowpass -> highpass). Refer to the code in **sbt.c** for potentially more clarification.

NOTE the fixed point precision for determining what block a pixel (x, y) lands in is **14**.

The function ***reflect*** is defined as follows:

```
reflect(i, n)
{
    if (i < 0) {
        i = -i;
    }
    if (i >= n) {
        i = n + n - i;
    }
    return i;
}
```

C.3.2.1 Inverse Transforms

The inverse 2D LL transform is defined as follows (gets applied to intra luma planes at levels 3 and 4):

```
for (every column)
  inverse_LL(column, HEIGHT, WIDTH)
for (every row)
  inverse_LL(row, WIDTH, 1)

inverse_LL(out, in, n, s)
  int even_n = n & ~1, h = n + (n & 1);
  for (i = 0; i < even_n; i += 2) {
      out[(i + 0) * s] = ((in[(i + 0) / 2 * s]) / 2);
      out[(i + 1) * s] = ((in[(i + h) / 2 * s]) / 2);
  }
  if (n & 1) {
      out[(n - 1) * s] = ((in[(n - 1) / 2 * s]) / 2);
  }
  out[0] -=  out[s] >> 1;
  for (i = 2; i < even_n; i += 2) {
      out[i * s] -= (-out[reflect(i - 3, n) * s] +
          5 * (out[(i - 1) * s] + out[(i + 1) * s]) -
               out[reflect(i + 3, n) * s] + 16) >> 5;
  }
  for (i = 1; i < n - 1; i += 2) {
    out[i * s] += (out[(i - 1) * s] +
                   out[(i + 1) * s] + 1) >> 1;
  }
  if (!(n & 1)) {
      out[(n - 1) * s] +=  out[(n - 2) * s];
  }
```

The inverse 2D L2 transform is defined as follows (gets applied to intra chroma planes at level 2):

```
for (every column)
  inverse_L2(column, HEIGHT, WIDTH)
for (every row)
  inverse_L2(row, WIDTH, 1)

inverse_L2(out, in, n, s)
  int even_n = n & ~1, h = n + (n & 1);
  for (i = 0; i < even_n; i += 2) {
      out[(i + 0) * s] = (2 * (in[(i + 0) / 2 * s]) / 5);
      out[(i + 1) * s] = ((in[(i + h) / 2 * s]) / 2);
  }
  if (n & 1) {
      out[(n - 1) * s] = (2 * (in[(n - 1) / 2 * s]) / 5);
  }
  out[0] += out[s] >> 1;
  for (i = 2; i < even_n; i += 2) {
      out[i * s] -= (out[(i - 1) * s] +
                     out[(i + 1) * s] + 2) >> 2;
  }
  for (i = 1; i < n - 1; i += 2) {
    out[i * s] += (out[(i - 1) * s] +
                   out[(i + 1) * s] + 1) >> 1;
  }
  if (!(n & 1)) {
      out[(n - 1) * s] +=  out[(n - 2) * s];
  }
```

The inverse 2D L2A (adaptive) transform is defined as follows (gets applied to intra luma planes at level 2):

```
for (every column)
  inverse_L2A(column, HEIGHT, WIDTH)
for (every row)
  inverse_L2A(row, WIDTH, 1)

inverse_L2A(out, in, n, s)
  int even_n = n & ~1, h = n + (n & 1);
  for (i = 0; i < even_n; i += 2) {
      out[(i + 0) * s] = (2 * (in[(i + 0) / 2 * s]) / 5);
      out[(i + 1) * s] = ((in[(i + h) / 2 * s]) / 2);
  }
  if (n & 1) {
      out[(n - 1) * s] = (2 * (in[(n - 1) / 2 * s]) / 5);
  }
  out[0] -=  out[s] >> 1;
  for (i = 2; i < even_n; i += 2) {
      if (IS_RINGING(i)) {
          out[i * s] -= (-out[reflect(i - 3, n) * s] +
          3 * (out[(i - 1) * s] + out[(i + 1) * s]) -
               out[reflect(i + 3, n) * s] + 2) >> 2;
      } else {
          out[i * s] -= (-out[reflect(i - 3, n) * s] +
          3 * (out[(i - 1) * s] + out[(i + 1) * s]) -
               out[reflect(i + 3, n) * s] + 4) >> 3;
      }
  }
  for (i = 1; i < n - 1; i += 2) {
    out[i * s] += (out[(i - 1) * s] +
                   out[(i + 1) * s] + 1) >> 1;
  }
  if (!(n & 1)) {
      out[(n - 1) * s] +=  out[(n - 2) * s];
  }
```

The inverse 2D L1 transform is defined as follows (gets applied to intra luma planes at level 1):

```
for (every column)
  inverse_L1(column, HEIGHT, WIDTH)
for (every row)
  inverse_L1(row, WIDTH, 1)

inverse_L1(out, in, n, s)
  int even_n = n & ~1, h = n + (n & 1);
  for (i = 0; i < even_n; i += 2) {
      out[(i + 0) * s] = ((in[(i + 0) / 2 * s]) / 2);
      out[(i + 1) * s] = (in[(i + h) / 2 * s]);
  }
  if (n & 1) {
      out[(n - 1) * s] = ((in[(n - 1) / 2 * s]) / 2);
  }
  out[0] -=  out[s] >> 1;
  for (i = 2; i < even_n; i += 2) {
      out[i * s] -= (-out[reflect(i - 3, n) * s] +
          17 * (out[(i - 1) * s] + out[(i + 1) * s]) -
              out[reflect(i + 3, n) * s] + (1 << (6 - 1))) >> 6;
  }
  for (i = 1; i < n - 3; i += 2) {
      out[i * s] += (-out[reflect(i - 3, n) * s] +
          5 * (out[(i - 1) * s] + out[(i + 1) * s]) -
              out[reflect(i + 3, n) * s] + (1 << (3 - 1))) >> 3;
  }
  while (i < n - 1) {
      out[i * s] += (out[(i - 1) * s] +
                      out[(i + 1) * s] + 1) >> 1;
      i += 2;
  }
  if (!(n & 1)) {
      out[(n - 1) * s] +=  out[(n - 2) * s];
  }
```

C.3.3 Subband Recomposition

Subband recomposition is achieved by the following pseudo code:

```
num_levels(width, height)
    max = MAX(width, height)
    lb2 = integer_log_base_2(max)
    if (max > (2 ^^ lb2)) { /* two to the power of lb2 */
       lb2 += 1
    }
    return lb2;

/* coefs = subband coefficients, w = width, h = height,
 * component = which color plane is to be recomposed,
 * Q = quantization parameter
 */
recompose(coefs, w, h, component, Q)
    levels = num_levels(w, h);
    HQP = get_HQP(Q);
    for (i = levels; i > 0; i--) {
      if (IS_INTRA AND IS_LUMA AND i == 1) {
          INVERSE_L1_2D(coefs, w, h);
      } else if (IS_INTRA AND IS_LUMA AND i == 2) {
          INVERSE_L2A_2D(coefs, w, h);
      } else if (IS_INTRA AND IS_NOT_LUMA AND i == 2) {
          INVERSE_L2_2D(coefs, w, h);
      } else if (IS_INTRA AND IS_LUMA AND (i == 3 OR i == 4)) {
          INVERSE_LL_2D(coefs, w, h);
      } else {
          if (IS_LUMA OR IS_INTRA) {
              INVERSE_HAAR_FILTERED(coefs, w, h, i, HQP);
          } else {
              INVERSE_HAAR_SIMPLE(coefs, w, h, i);
          }
      }
    }
    for (all recomposed pixels)
       add 128 to pixel and CLAMP it to [0, 255]
```

D. MOTION COMPENSATION

Motion compensation is done per plane. Do not expand the bit depth of the image at any point to do motion compensation.

D.1 Compensating Inter Blocks

Inter blocks are compensated using half or quarter-pixel precision. The luma plane has a special filter for computing the subpixels and the chroma planes use simple bilinear filtering.
Motion vectors for chroma are derived (in the case of chroma subsampled formats) by dividing the x and/or y component of the motion vector by the ratio of chroma subsampling in that direction. For example, in a 4:2:0 scheme, both the x and y components are divided by two for the chroma planes.

Luma Half-Pixel Filter (Bicubic)

```
(19 * (b + c) - 3 * (a + d) + 16) / 32
```

Luma Quarter-Pixel Filter and Chroma Half-Pixel Filter (Bilinear)

```
(a + b + 1) / 2
```

Compute the filtered block and add it to the current block. Compensate according to **D.4**.

D.2 Compensating Intra Blocks

For each of the four intra sub-blocks, calculate the average value of the sub-block in the reference picture at the same location and add it to the pixels in the current sub-block. If the block is not intra, add the reference sub-block directly.
Compensate according to **D.4**.

D.3 Caveat For Encoder

From DSV1: ~~Due to the restriction of 8 bits per channel, the compensation cannot account for intra blocks that have large changes. Therefore, the encoder must simulate the reduced range intra block motion compensation to see if the block can be successfully reconstructed before it decides to label the block as intra.~~

In DSV2, this limitation was removed by introducing EPRM motion compensation. The encoder, however, should still do its own analysis to decide whether the block needs to be marked as EPRM to effectively predict and compensate the difference.

D.4 Reconstruction

If block is not flagged to use Expanded Prediction Range Mode (EPRM) or the block is a skipped inter block, reconstruction of a pixel is as follows:

```
CLAMP(prediction + residual - 128, 0, 255)
```

Otherwise, if the block is flagged to use EPRM, this is how its pixels are reconstructed:

```
CLAMP(prediction + (residual - 128) * 2, 0, 255)
```

D.5 In-Loop Filtering

A brief overview of the filtering done to the motion compensated frame. All planes are filtered, not just luma. Refer to the code in **bmc.c** for potentially more clarification.

Intra blocks and non-skipped inter blocks get filtered.

D.5.1 Filter Thresholds

Intra block filter threshold is computed as follows:

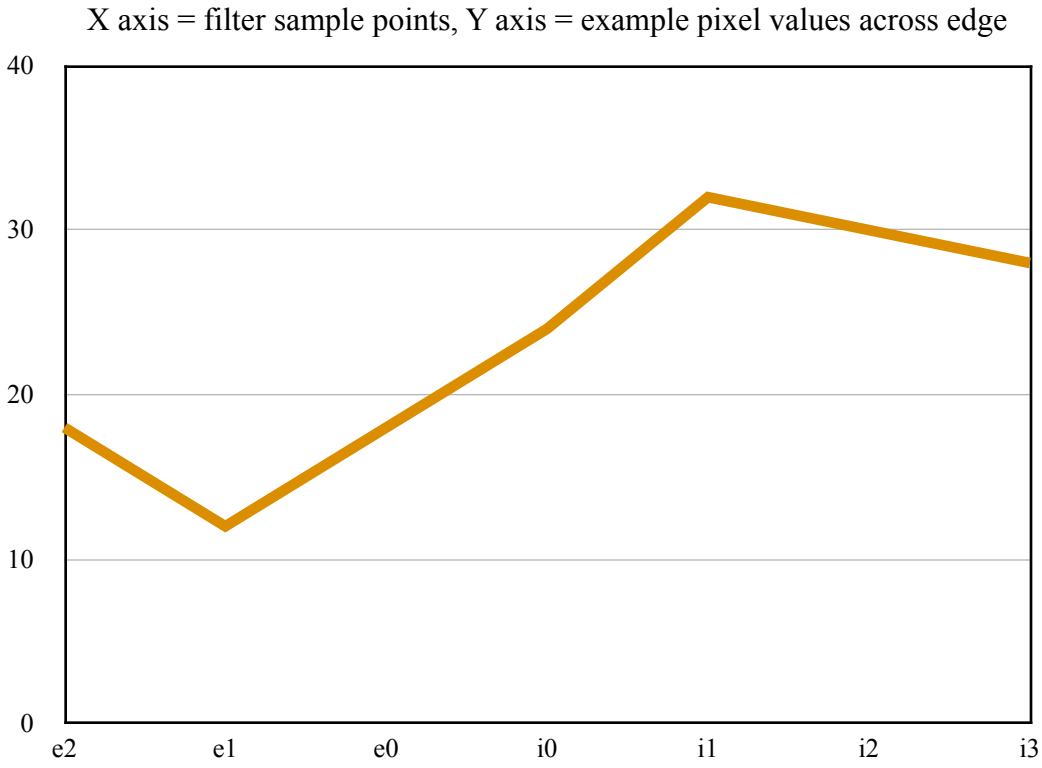
```
CLAMP((64 * Q) / 4096, 2, 24)
```

Inter block filter threshold has a more sophisticated computation. For a given block, it looks at the motion vectors of the block's left and top neighbors and determines how similar its own motion vector is to them. Pseudocode for the computation looks like this:

```
est_mag2(ix, iy)
   ix = absolute_value(ix)
   iy = absolute_value(iy)
   if (ix > iy) return (3 * ix + iy) / 4
   return (3 * iy + ix) / 4

difsq2(v0, v1)
   if ((v0 IS ZERO) || (v1 IS ZERO)) return 0
   return est_mag2(v0.x - v1.x, v0.y - v1.y)


CMV = THIS_BLOCK.MV /* current motion vector */
LMV = LEFT_BLOCK.MV
TMV = TOP_BLOCK.MV

if (LEFT_BLOCK DOES NOT EXIST OR IS NOT INTER)LMV = CMV
if (TOP_BLOCK DOES NOT EXIST OR IS NOT INTER) TMV = CMV

d0 = DIFSQ2(LMV, CMV)
d1 = DIFSQ2(TMV, CMV)

minD = MIN(d0, d1)
maxD = MAX(d0, d1)

if (minD > 0) return (16 * maxD * Q) / 4096
return ((4 + 8 * (maxD - minD)) * Q) / 4096
```

D.5.2 Filtering

Filtering is done horizontally (top to bottom) across the vertical edges (left and right) first, then it is performed vertically (left to right) across the horizontal edges (top and bottom). Pixels on the outer edge of the video frame are not filtered.



Sample points for the filtering are prefixed with ‘e’ (external) or ‘i’ (internal). External points are outside of the current block while internal points are inside or on the edge of the current block.

```
TESTINTRA_1 INTRA && (abs(e1 - i1) < thresh)
TESTINTRA_2 (abs(e1 - avg) < thresh &&
             abs(e2 - avg) < thresh &&
             abs(i1 - avg) < thresh &&
             abs(i2 - avg) < thresh)

TESTINTER_1 INTER && (abs(i0 - e0) < ntt)
TESTINTER_2 abs((i0 - i1) * 4) < ntt && abs((e0 - e1) * 4) < ntt
TESTINTER_3 abs(i0 - i1) < ntt && abs(e0 - e1) < ntt
```

Filtering conditions are given above, ***abs*** being absolute value. The variable ***ntt*** is computed like this:

```
compute_ntt(threshold)
{
    threshold = MIN(threshold, 5);
    return threshold * threshold;
}
```

The ***o_*** prefix represents the pixel at that sample point, used to designate which sample point is getting updated by the filtered value. The logic is as follows:

```
if (TESTINTRA_1 && TESTINTRA_2) {
   o_e1 = (avg + e1 + 1) / 2;
   o_e0 = avg;
   o_i0 = avg;
   o_i1 = (avg + i1 + 1) / 2;
} else if (TESTINTER_1) {
   if (TESTINTER_2) {
      i0 = (e0 + i0 + e1 + i1 + 2) / 4;
      o_i0 = ((i2 + (i1 * 2) + (i0 * 4) + e1 + 4) / 8);
      o_i1 = ((i2 + i1 + (i0 * 2) + 2) / 4);
      o_i2 = (((i3 * 2) + (i2 * 3) + i1 + (i0 * 2) + 4) / 8);
   } else {
      if (TESTINTER_3) {
         o_e1 = ((e2 + e1 + e0 + i0 + 2) / 4);
         o_e0 = ((e2 + e1 * 2 + e0 * 2 + i0 * 2 + i1 + 4) / 8);
         o_i0 = ((i2 + i1 * 2 + i0 * 2 + e0 * 2 + e1 + 4) / 8);
         o_i1 = ((i2 + i1 + i0 + e0 + 2) / 4);
      }
   }
}
```

For the opposite edge, invert the sample points (i becomes e, e becomes i).