Intel® Stress Bitstreams and Encoder (Intel® SBE) 2016 – VP9 User Guide

Version 2.2
Updated March 28, 2016

Contents

Introduction

Testing a decoder is a complex multi-criteria task. Code coverage of 100% lines of a decoder code does not guarantee the decoder is 100% compliant. At the same time, creation of millions of streams to test all possible feature combinations is time and storage consuming. Random Encoder partially resolves these two issues. It is a lightweight encoder with no mode decision, so it is as fast as the decoder is. Effective motion estimation and mode decision are not parts of video codec standard and are not required to be tested at the decoder side. So this most time-consuming part is mostly omitted in Random Encoder in favor of speed and flexibility.

Codec developer will not want to keep all streams generated by random encoder, it is enough to keep only basic vectors. If a decoder fails to correctly decode a randomly generated stream, it makes sense to extend the test pool with the stream for future regression validation. Random Encoder is a great extension of codec validation in addition to basic vectors provided with Stress Bitstreams.

Feature Sets and Profiles

VP9* specification has two feature sets: Feature Set 1 (FC1) containing only 8-bit 4:2:0 colorspace (chroma components reduced in both directions) and Feature Set 2 (FC2) allowing 10 and 12-bit color depth and more flexible options of chroma subsampling.

Profile is a syntax element in frame header defining color bit depth and chroma subsampling setting scope:

- Profile 0 allows only 8-bit 4:2:0 color representation.
- Profile 1 adds an option to disable chroma subsampling in vertical, horizontal or both directions.
- Profile 2 enables 10- or 12-bit color depth with 4:2:0 color representation.
- Profile 3 enables both high-bit color depth and chroma subsampling options.

Random Encoder supports all bit-depth options and chroma-subsampling configurations. All the information in this document is applicable to any Profile and Feature Set. For more information about high-bit depth and chroma subsampling, see *VP9* section.

General Overview

Random Encoder is a highly configurable and flexible syntax (VP9) encoder tool. Unlike regular encoders, it is not intended to achieve compression but only designed to create a valid specification-compliant stream. Compliant streams contain only allowed combinations of syntax elements and their values to test decoder for unusual cases or boundary stress cases where developers usually relax requirements to code development in favor of higher decoding speed. Decoder must be compatible with any stream so its sloppy optimizations have to be carefully tracked for boundary cases where residuals overflow may break visual representation of the picture.

You can find the recommended Decode validation process with Random Encoder below. It is up to the user to change the flow and to react on pass and fail events and even decide on the

criteria of test passing. We recommend to use the latest development branch of libvpx reference decoder for VP9 for testing.

As an input, Random Encoder accepts *an optional YUV file* and a *parfile* describing testing settings: features to utilize, fixed values, random values. As an output, Random Encoder produces an *encoded bitstream* and optionally writes a *YUV file* with internal reconstruction data. This file is used to validate that Encoder generated a proper compressed file and that the resulted bitstream is valid.

If there is a mismatch between encoder reconstruct and reference decoder result, you are welcome to report this to Software Publisher (your Intel contact) with the case configuration to request the fix, if it fits your license agreement with Intel. We will always appreciate your feedback.

Random Encoder has the *seed* parameter (-s) defining initial random-engine state. If you change it, you can use the same parfile to produce totally different streams with the same scope of randomization defined by parfile. The main purpose of this seed feature is extensive testing with all possible syntax-element combinations. Besides, this feature can be used to create small bug-reproducers (setting frame number parameter "-f" to some small value) for the parfiles which are known to generate the streams causing failure of the examined decoder.

To summarize, a typical workflow of compliance testing for VP9 consists of the following steps:

1. Produce test stream by feeding Random Encoder with a parfile and an optional input YUV file.
2. Decode the stream with reference decoder.
3. Verify that Encoder's reconstruct matches the reference decoder result.
4. Decode the stream with your decoder and verify that its result matches the reference decoder result.
5. Increment the *seed* parameter and return to step 1.

VP9 Specifics

To start validation cycle with Random Encoder for VP9, make a decision regarding your testing agenda and set up certain features and value range in random-encode parfile. Parfile is a JSON formatted file that contains the settings for all syntax elements and some additional entries related to frame scaling and residual randomization. Wide range of values in the parfile will result in a very detailed testing. Random Encoder accepts a YUV file and a parfile and iterates each time the "--seed" command-line option changes, producing a new output stream for each iteration.

Here is a simple example when it is reasonable to narrow the scope of search. If you want to check how the decoder handles a whole range of motion vectors, then set the "mv_len" parameter in the "superblock" section of the parfile to a maximum value. Magnitude of each MV component will be uniformly distributed in range [−mv_len..mv_len]. But if you need to test interpolation filters and convolution implementation, you need to ensure that 1/8-pixel MV accuracy is utilized, and it is considered only if neighbor motion-vector components have the magnitude not greater than 8 pixels. For proper testing, you need to set "mv_len" to value 64

(see the detailed description of this parameter explaining why all values are multiplied by 8), this will provide uniform distribution between all convolution vectors of each interpolation filter. Otherwise, the chances to produce 1/8-pixel accurate MV are very small, so proper testing of convolution implementation will take a long time.

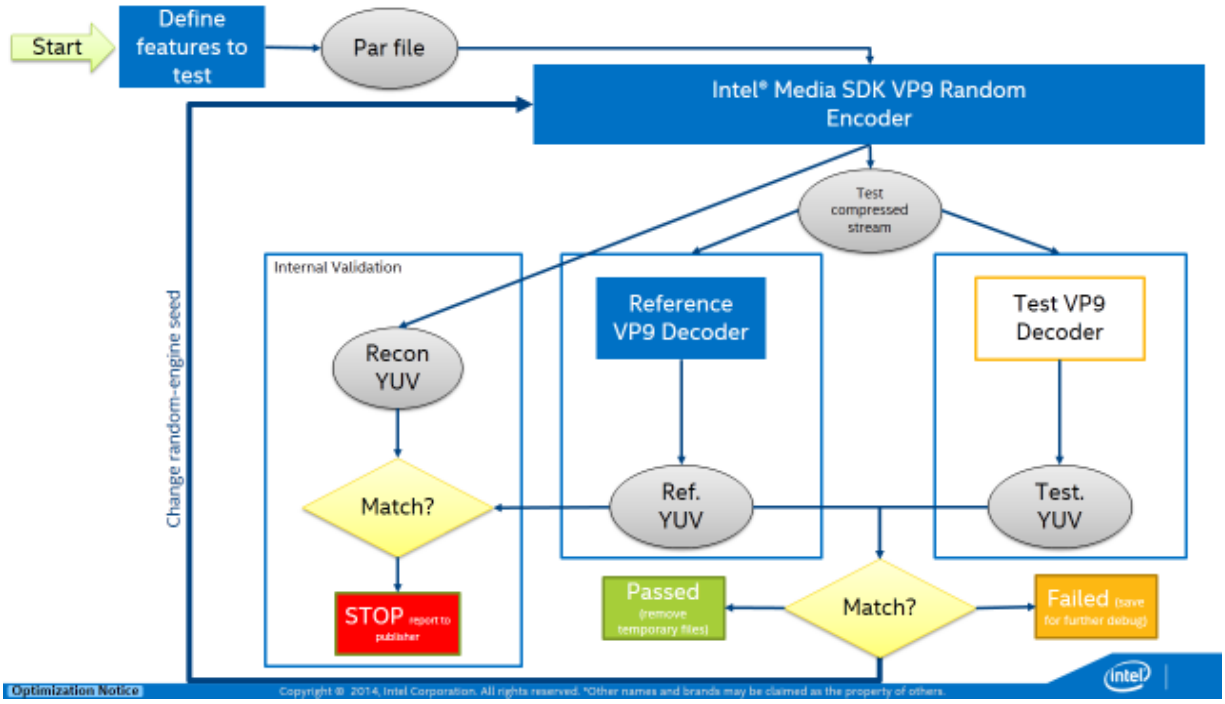


Figure 1. Testing workflow for Random Encoder for VP9.
The scope of testing is defined in a parfile. Encoder produces a compressed bitstream with randomized mode decisions and a YUV file with internal reconstruct (optional). To validate your decoder, compare its results of randomized bitstream decompression against the results of the reference decoder. The purpose of the YUV file with reconstruct is validation of the Random Encoder—it also must match the reference decoder result.

Parfile Editing Basics: How to Setup Test Coverage

This chapter explains in detail the format that was chosen for test coverage.

Let's start with a basic question: why do we need a parfile? Random Encoder makes each decision randomly, there are no constraints or behavior patterns. You can expect any possible syntax-element combination from the Encoder, as long as this combination is compliant to the VP9 format. Two possible problems arise here.

The first problem is that we may not want *any* combination: for example, we are at an early development stage and don't have all the features implemented, so it doesn't make sense to test those features.

The second problem is test-coverage control. For example, Encoder has to choose the current frame type: key or Inter. We could give equal chances for both variants, or put key frame about every second or every 10 seconds. Equal chances may not be good for inter-prediction feature coverage, since inter-prediction features may form more complicated feature combinations and

require more detailed testing; 10 seconds may be too rare for accurate key frame testing, intra-prediction testing and key/inter frame combinations; one second may seem fair but it still doesn't look like a flexible enough solution. Besides, we need a possibility to disable key frames for extensive testing of intra prediction.

The solution we came with is definition of chances for certain values of enumerated syntax element by probability weights. Frame type has two options: Key and Inter. To have equal chances for both, we add the following line into the parfile: "frame_type" : [1, 1]. If we want to have a key frame about every second, we increase the chances of Inter frame: "frame_type" : [1, 29]. This defines the chances of getting a key frame as 1/30, and, what is important, doesn't define any strict pattern allowing to produce any possible combination of key and inter frames. Sometimes it is even possible to have two consecutive key frames, but the average ratio of key-frame number to the total length will be about 1/30 for the long enough streams. And if we want to disable inter frames we just give them zero chance: "frame_type" : [1, 0]. The solution is quite flexible and is applicable to all enumerated syntax elements. Most of the elements are named the same or similar way as variables in libvpx source code (reference decoder VP9 implementation).

Some syntax elements would be unreasonable to enumerate, for example, *base_qindex* has the range of 256 acceptable values. Instead, you would typically prefer one of the following options:

- fixed value
- range of small or large QP values
- full range
- full range except zero, which may turn lossless mode on

Such range parameters are defined in the parfile with minimum and maximum values, so random values will be distributed uniformly in this range. For example, the full range of base_qindex is defined as follows: "base_qindex_range" : [0, 255]. If we want to exclude zero value, we write "base_qindex_range" : [1, 255]. To make QP fixed, for example, equal to 40, we set min=max: "base_qindex_range" : [40, 40].

JSON syntax was chosen since it is convenient both for reading and for manual editing, but initially, this format also had some drawbacks. There are no syntax cues to recognize weight-based parameters from range-based ones, and for weight-based parameters the order of enumeration may be unobvious. These drawbacks are compensated by the comments for each parameter prompting the type of the parameter, the acceptable values for range-based parameters, and the actual enumeration values of weight-based parameters. Parameters not corresponding to VP9 syntax-elements and related to Encoder's behavior control have comments with additional explanations.

Such JSON definition extended with comments turned out to be a good tradeoff between flexibility of test-coverage setup, human readability and compatibility with automatic parsers.

Parfile Editing: a Short Summary

A parfile is a file in JSON format with comments. It includes the following general sections related to different groups of parameters:

- The "frame" and "superblock" sections allow to control syntax-element randomization scope in the frame header and at the superblock level. Most elements in these two sections are named according to the corresponding variables in libvpx source code.
- The "scaling" section defines how frame resolution will be varied in the encoded stream.
- The "residual" section contains the settings related to residual randomization at the superblock level.
- The "override" section allows to redefine the settings for the specified frames.

There are two types of parameters: weights and range. In most cases logical or enumerated parameters are defined by relative weights. For example, setting "frame_type" : [1, 10] means one chance of key frame against 10 chances of inter frame. Numeric parameters with wide range are defined by specifying minimum and maximum values. For example, for the loop filter level, acceptable values range from 0 to 63, and in parfile it is defined as "lf_filter_level" : [0, 63]. Actual values will be spread uniformly in this range.

To disable randomization for weights-defined parameters, weights for all except one values must be set to zero. To set a range-defined parameter to a fixed value, you should set minimum and maximum values to the same value.

Description of Command-Line Application Options

Random Encoder is a command-line application that accepts the following parameters:

```
vp9_random_encoder.exe -i <input.yuv> -o <output.vp9> -w <width> -h <height>
                       [-p <config.json> -s <seed> -r <reconstruct.yuv>]
                       [-f <# of frames to process>]
```

Parameter description:

Parameter	Short description	Details
`-i STRING, --input=STRING`	Input file (raw YUV)	File path, currently only I420 YUV is accepted
`-o STRING, --output=STRING`	Output VP9 bistream	File is output in IVF container
`-w INT –h INT, --width=INT --height=INT`	Width and height of input YUV	
`f INT, --frames=INT`	Number of frames to process	Number of frames which random encoder will consume and encode. Decoding result may have different number of frames because some of encoded frames may be hidden and also bitstream may contain one-byte-repeated frames. If hidden frames and one-byte frame repeats are disabled, output will contain the same number of frames.
`-r STRING, --recon=STRING`	Output YUV file with encoder's reconstruct (optional)	Used to validate random encoder. If not set, encoder’s reconstruct won’t be written.
`-p STRING, --parfile=STRING`	JSON file with distribution parameters (optional)	Parfile setting limitations on syntax element randomization. If not set, the default values, which

		actually are not very practical, will be used.
`-s INT,` `--seed=INT`	Seed for random engine (optional)	Acceptable range: 0..INT_MAX
`--dump_hidden=BOOL`	Write invisible frames to reconstruct file (optional)	0 or 1, the default is 0. If is set to 1, encoder will output hidden frames to reconstruct. This option was added to enable use of libvpx reference decoder without modification, because there is no option to dump hidden frames. On the other hand, for thorough validation, you need to modify reference decoder and use output containing hidden frames.
`--output_state=STRING`	Path to the file to which states of random engine at keyframes are written (optional)	This is another way to get short-reproducer stream, but it is not well designed and needs to be accurate. You are recommended to avoid using this option and to iterate with *seed* instead. If you use this option to provide the file path, the encoder will dump the whole state of internal Mersenne-twister random-engine at each key frame. The state is a single line with a lot of numbers. If you want to start from the state of a certain frame, you can copy the corresponding line (without frame number) to a separate file and provide the path to this file using the --input_state option. Since the random encoder mostly doesn't rely on input frames for mode decision, there are very good chances to get the same combination of modes at the beginning of the bitstream, not somewhere in the middle. However, there are much better ways to get short reproducers. In addition to iterating with seed, you can just cut a certain part with Intel® Video Pro Analyzer (Intel® VPA) or any other tool that can parse and edit IVF format.
`--input_state=STRING`	Path to input file with random engine state (optional)	See --output_state option description.
`--bit_depth=INT`	Bitstream bit depth (8, 10, 12), overwrites parfile setting	Only for the version with FC2 support, see *VP9* section for details.
`--recon_bit_depth=INT`	Encoder-reconstruct bit depth (8, 10, 12, 16), for easier comparison with reference decoder results	Only for the version with FC2 support, see *VP9* section for details.
`--colorspace=STRING`	Bitstream colorspace: yuv420|yuv422|yuv440|yuv444, overwrites parfile setting	Only for the version with FC2 support, see *VP9* section for details.
`-t STRING,` `--target=STRING`	Target mode: random | visual | mem_bw • random (default) - no restriction, full randomization • visual - try to avoid some configurations yielding visual artifacts • mem_bw - optimize for memory bandwidth testing	At the current stage, visual mode does two things: • Repeats each invisible frame with s*how_existing_frame* flag. This allows to make playback smooth without disabling invisible frames in the parfile. • Disables residual randomization. All other settings potentially yielding visual artifacts are to be controlled through the parfile. "mem_bw" (memory_bandwidth) mode enforces reference frame sign bias flags to have different values, so compound prediction is available at each inter frame. All other settings recommended for memory bandwidth testing are provided in 401th parfile
`--allow_idct_overflow=BOOL`	Disable check that randomized	Residual randomization in combination with high QP may

	residual doesn't yield inverse-transform overflow	produce illegal combination of coefficients yielding inverse transform overflow. During residual randomization, Encoder performs additional checks dropping illegal combinations. To disable these checks, set this option to 1.
`--extreme_residual=BOOL`	Use only extreme values for random residual	Effective if residual randomization is enabled in parfile, helps to hit inverse-transform overflow.
`--idct_overflow_hw=BOOL`	Enable Emulate-Hardware-Highbitdepth behavior for iDCT calculation	Affects inverse transform, copies behavior of corresponding configuration define in libvpx code.
`--verbose`	Print reference frames' resolution for each frame (optional)	An option for debugging purposes. This option helps to understand dependencies between reference frames when you are working with a certain stream and parfile while frame scaling is enabled.
`--help`	Output help message	

Case Study: Focus on ADST Implementation

Let's assume you have optimized inverse Asymmetric Discrete Sine Transform (ADST) computation and want to validate the new implementation against the reference one. ADST is applied only to intra blocks, but for the default config the portion of key frames is 10% and the portion of intra blocks in inter frames is 25%. Taking into account that not all intra modes utilize ADST, about 75% of bitstream content will be irrelevant to the validation task.

It will be reasonable not to use inter frames for ADST testing. So *008_intra_stress.json* parfile is a good starting point, and it will be modified to setup Random Encoder for ADST testing. First of all, let's perform smoke testing using this parfile without modification on a small number of frames (about 20 at SD resolution). If it passes, you can move ahead. If not, you can check that each size of transform works correctly by creating separate parfiles for each transform modifying the following line in the "frame" section:

```
"tx_mode" : [1, 1, 1, 1, 2], // 4x4, 8x8, 16x16, 32x32, select
```

Let's disable all transform except 4×4:

```
"tx_mode" : [1, 0, 0, 0, 0], // 4x4, 8x8, 16x16, 32x32, select
```

Then do the same for 8×8:

```
"tx_mode" : [0, 1, 0, 0, 0], // 4x4, 8x8, 16x16, 32x32, select
```

And the same way for 16×16. In fact, when you set "tx_mode" to enable 8x8 transforms only, this setting doesn't fully disable 4×4 transforms, they are still used for superblocks less than 8×8 and for chroma in blocks less than 16×16. If it is necessary to fully disable small transforms, modify the "partition" setting in "superblock" section to disable superblock splitting at a certain level. If there were any basic problems with the new ADST implementation, they would be revealed by such basic tests.

Now let's focus on validation of extreme cases. Since it is important to check extreme values, increase the portion of blocks with randomized residual up to 50% by changing the following line in the "residual" section:

```
"sb_randomized"  : [9, 1], // no | yes
```

to

```
"sb_randomized"  : [1, 1], // no | yes
```

The original version defines chances that superblock has random residual as one against nine, which yields only 10% of blocks with random residual. In the modified version, the changes are one-to-one, encoder will produce 50% of such blocks.

The "tx_mode" parameter described above should be set to allow 4×4, 8×8 and 16×16 transforms with equal chances (32×32 blocks are processed only with DCT):

```
"tx_mode" : [1, 1, 1, 0, 0], // 4x4, 8x8, 16x16, 32x32, select
```

In addition, you may disable intra-prediction mode, which don't utilize ADST (DC and 45°) by changing the "intra_mode" setting in the "superblock" section as follows:

```
//                  //DC,  V,  H, 45,135,117,153,207, 63, TM
"intra_mode"        : [0,  1,  1,  0,  1,  1,  1,  1,  1,  1],
```

Summary: to validate the ADST implementation, we selected the parfile without inter frames as the starting point; the portion of blocks with random residual was increased in order to test inverse transform for handling extreme values; 32×32 transform size and transform-size definition at the superblock level were disabled. Besides, the intra modes not utilizing ADST were disabled. This configuration is dedicated to validation of ADST transform, and if there is a possible mismatch with reference implementation, the chances that it will be revealed are several times higher than those for the default all-enabled configuration are.

How to Choose Input Resolution

Recommended resolution for input video is 432×240 or close to it. This is a trade-off between performance and test coverage. Globally, we have two separate scopes of value randomization: frame header and superblock elements. Larger resolution (for example, Full HD) will lead to lower Encoder speed in terms of FPS, which means that we will write fewer frames and will try fewer frame-header element combinations in the same period of time. For superblock-element combination coverage, resolution does not matter much. However, using the resolution smaller than 432×240 will yield a video with fewer superblocks, which will significantly reduce coverage of dependencies between neighbor superblocks.

Some features need input video of large resolution for proper testing. For example, maximum magnitude of MV component is limited by resolution and is not reached at 432×240, so for testing MV extreme values, Full HD input is recommended. Another example is the number of vertical tiles. Minimum tile width is 256 pixels, so at bitstreams with frame width less than 512 pixels, vertical tiles can't appear at all. It's recommended to test tiles on 4K+ content and also try frame width not divisible by 256.

Another specific feature is frame scaling. First of all, it is recommended to perform separate validation of frame scaling and other features. The main reason is that scaling complicates the debugging process. Besides, for frame-scaling test you need an input of quite a large resolution to test the edge cases of different reference-frame resolution: for proper testing of all combinations, it's recommended to have an input file of dimensions that produce reasonable values after division by 32 (for example, 4096×2048 is OK).

Summary: basic recommended resolution is 432×240, it is a good trade-off between performance and frame-header feature coverage; some cases require Full HD or larger input resolution; frame scaling also requires large input resolution and we recommend to test other features thoroughly prior to frame-scaling validation.

How to Control Frame-Scaling Randomization

The "scaling" section is the most unobvious one as compared to other parfile sections, but it is quite important because on-the-fly frame scaling is a new feature in VP9 and it requires special attention.

This parameter section provides enough flexibility to generate basic scaling cases, corner cases and extensive-testing streams with arbitrary resolution. VP9 has a limitation on reference scaling factor (2x downscale and 16x upscale), so the parameters don't control absolute values in pixels, only relative factors.

Random Encoder doesn't upscale the input frames, only downscales them. So the maximum resolution is defined by the resolution of input video. The minimum frame size can be limited with the "min_frame_size" parameter. Each frame is scaled independently of other frames, but 0.5x−16x ref scaling limit is taken into account.

The "resolution_change" parameter enables frame scaling. It is a weight-based parameter that defines the chances for the current frame to be scaled.

Three similar parameters are defined by pairs of floats: "intra_max_downscale", "max_upscale_factor" and "max_downscale_factor". The first number defines max scale factor for frame width, the second one defines max scale factor for frame height. These parameters are applied in different cases. "intra_max_downscale" applies to key frames and intra-only frames; these types of frames are free from restrictions on reference frame size, so intra frames are allowed to be any size. Other two parameters apply to inter frame scaling.

Resolution randomization of inter frames is performed as follows:

1. Encoder picks one reference frame with ref frame index from the range defined by the "base_ref_frame" parameter.
2. For each dimension, Encoder decides whether to increase or decrease it by chances from "downscale_or_upscale".
3. Encoder picks a random value in the range between the original size and the size defined by max upscale/downscale factor. Minimum allowed value is 4, maximum is limited by input video resolution.

There is an extra parameter "scale_first_frame" that defines whether the first frame is downscaled. This enables the decoder to start with the largest frame size and thus allocate enough memory. Don’t expect such behavior in real world, but feel free to use it for debugging purposes. This parameter is defined by string, possible values are "no", "yes" or "maybe". The "maybe" option leaves to decoder the decision whether to scale the first frame for the current stream or not.

In order to specify exact dimensions, parameter "frame_size_px" is used.

Extensive Testing and Parallel Execution

Random Encoder completes as many as ≈50 000 cases a week for a single process. For satisfactory validation, it needs to run for a month with NO fails in decoder on a corner case. To increase coverage for a shorter time period, you can execute several processes in parallel on a single system, as many as hardware memory and CPU cores allow.

You may split parallel tasks by parfiles, input resolutions and seed interval. It is reasonable to spend most of CPU time testing all-enabled stress parfile at a small resolution, but also have some additional launches focusing on the following targets:

- Large resolutions, tiles and long motion vectors
- Interpolation filters, motion vectors ≤ 8 pixels, randomized residual to test inverse transform and convolution for extreme cases
- Frame scaling with a very large range of possible resolutions

Using VP9 Random Encoder as a Debugging Tool

In addition to stress-testing after basic validation, VP9 Random Encoder is a useful debugging tool, which can help you identify the exact feature that got broken after changes in the decoder.

Starting from the 1.2.0 release, Intel® SBE includes the parfiles used for Stress Bitstreams generation. Parfiles contain the same information as the spreadsheet defining the syntax-element utilization, but they are more convenient for debugging purposes as you can feed them into the Encoder without any modifications.

The Stress Bitstreams are organized in such a way that their complexity and the number of utilized features increase with the stream index. So if several of your streams failed validation, you should start debugging with the one having the lowest stream index value.

The proposed workflow for debugging with Intel® SBE is as follows:

1. Pick the failed stream with the lowest stream-index value and the stream with the preceding stream-index value that passes validation.
2. Pick the corresponding parfiles and generate new streams with Random Encoder. Ensure that behavior of the tested decoder on the generated streams is the same as on the original ones.

3. Compare “good” and “bad” parfiles side-by-side in a text comparison tool to identify the feature difference. Most of the parfile lines are mapped directly to VP9 syntax elements. Changes between consecutive parfiles include several syntax elements, usually related to one feature or similar features.
4. Try to disable the identified new features one by one, to determine whether they affect the decoder behavior. In some cases, this information combined with knowledge about recent decoder modifications will be enough to figure out the cause of the failure.
5. Narrow the scope of possible values for problem syntax element.
   When the problem syntax element is defined, sometimes it is possible to narrow the scope of utilized values. For example, if a decoder produces wrong output on streams generated by Random Encoder, but behaves properly if you disable sharp interpolation filter, the problem is most likely with the handling of sharp interpolation filter.
6. Continue simplification of the “bad” parfile by disabling other features.
   This step pursues two goals: check cross-feature dependencies and simplify further debugging.
   a. To illustrate the first goal, *intra_only* flag in header causes setup_past_independence() function call, which resets some of decoder contexts and buffers. Disabling *intra_only* element in parfile checks whether this case affects the considered feature of “bad” parfile. Other noticeable features to be tested first for cross-dependence: frame resolution change, *show_frame* flag (frame visibility), segmentation, tiling, frame context operation flags.
   b. Even if all this stuff didn’t help, it would be easier to go into deeper debugging if we have the stream with only the largest partitions (simpler call stack and debugging path) and the smallest transforms (easier to observe the values of the whole block). Of course, if the bug still occurs on such a configuration. You can also try to set max motion vector length to 0, leave only one interpolation filter, disable intra prediction on intra frames, disable compound prediction and so on.
7. Iterate with the “--seed” option to find a shorter reproducer.
   Sometime the bug is occurring in quite rare cases and far from the beginning of the stream. To get a shorter reproducer, limit the length of the bitstream produced by random encoder by some small value and iterate with the *seed* parameter value, which controls the initial state of random engine. For intra frame bugs, a single frame will usually be enough. For inter-prediction bugs, we recommend to do the reduction of stream length in several steps, because in the case of specific dependency between several frames 2-frames reproducer may not be enough.

Smoke Test Streams

In addition to the default streams, Intel® SBE includes a small set of streams for fast smoke testing. This set has the same reference code coverage as the default set, and allows to perform a much faster test and validation cycle. Stream set has a much smaller memory footprint in comparison to the default set. As a drawback, it has degradation in cross-coverage of multiple syntax elements.

VP9 FC2 Specifics

The Random Encoder supports all Profiles of VP9 format: color depth of 8, 10 and 12 bits with different settings of chroma subsampling. To enable high-bit depth encoding, use `--bit_depth` and `--colorspace` command-line options or the "color" section in parfile. When command-line and parfile settings are used together, the command-line settings have a priority. Generation of bitstreams with mixed bit depth or colorspace is possible only with parfile color setting. Supported values for `--bit_depth` option are `8`, `10` and `12`; for `--colorspace` option: `yuv420`, `yuv422`, `yuv440`, `yuv444`.

The Random Encoder supports only 8-bit YUV 4:2:0 input and performs necessary color conversions internally. Bit-depth conversion is performed as shifting the 8-bit value by 2 or 4 bits. The lower bits are randomized. For chroma upscale, the Encoder uses nearest-neighbor method.

To simplify validation and debugging, you can use the `--recon_bit_depth` option defining bit depth of YUV file with decoders reconstruct. Supported values are 8, 10, 12 and 16. libvpx decoder has a similar option `--output-bit-depth`. It may be useful to decode all high-bit-depth enabled streams to 16 bits for easier comparison against the reference decoder. Pay special attention to streams with mixed bit depth: the reference decoder defines output parameters as the first-frame parameters, and information loss is possible. The reference decoder does not have any options to unify output chroma subsampling format, so neither has the Random Encoder.

Profile, bit depth and subsampling randomization settings can be defined in the "color" section of the JSON parfile. All the parameters in the "color" section are weight-based. If you use --bit_depth or `--colorspace` command-line options, the "color" section in the parfile is ignored.

VP9 format allows profile, bit depth and subsampling settings to change only at key and intra-only frames.

Residual Randomization

To stress-test residual decoding, Random Encoder provides two modes of residual randomization: randomized residual and randomized transform coefficients. The mode is defined with the "type" parameter in the "residual" section, the available values are:

- "off" - no residual randomization
- "diff" - plain residual
- "transform" - transform coefficients

For each superblock, Random Encoder makes a decision whether to fill it with random residual or not. You can control the portion of such superblocks with the "sb_randomized" parameter.

In randomized-residual mode, randomization is performed independently for each pixel of the superblock. Pixel-residual value is picked from the range defined by the "residual_range" parameter (–255..255 by default). Please take into account that for 10- and 12-bit color depth this range is adapted by the encoder internally, no adjustments are required. So if you need

max range for 10-bit color, parameter value still needs to be [-255, 255] and Random Encoder will interpret it as [-255..255] for 8-bit frames and as [-1023..1023] for 10-bit frames.

Transform-coefficient randomization is a bit more complicated. Not all the coefficient configurations will have valid back-projection, so the encoder should be accurate in performing randomization. First, it picks end-of-block (EOB) token position. EOB token means that all coefficients starting from this position are equal to zero. Then the encoder picks a random position in the range from DC coefficient to EOB and puts in this position a random value according to coefficient position, transform size and type. If the result is a valid transform (inverse transform yields residual in acceptable range), the encoder tries to put one more coefficient. When encoder finally fails to get a valid transform, it steps one iteration back and stops. Depending on the "type" parameter in the "residual" section, coefficient values are generated either uniformly between tokens ("tokens") or in a way to yield uniform distribution of the most significant bit ("transform"), producing a reasonable amount of extreme-value cases.

Another option related to transform randomization is "tx_zero_coeff_tail". Its purpose is to deliver coverage of the following corner case. In a transform block, the last coefficient can't be zero, EOB token must always be preceded with a nonzero coefficient value. But if the block includes only N×N coefficients, EOB token is omitted. A block filled with the tail of zero coefficients instead of a single EOB token will be valid for VP9 decoder, and the "tx_zero_coeff_tail" parameter is responsible for chances of getting such block. The parameter is only applicable to blocks with randomized residual.

Residual and transform-coefficient randomization modes cannot be used simultaneously in a single stream. Besides, you should take into account that enabling any mode of residual randomization increases memory footprint in propotion to the size of input frame, in pixels.

What VP9 Random Encoder Does Not Support

- Superframes. This feature is related to IVF container, not to VP9 format. You can find streams for testing superframes in the publicly available libvpx test stream pool.
- display_width and display_height that differ from source-frame size. Frame header may have a special flag in the parfile. When set to 1, this flag indicates that the display size is different, but the actual display size values will be equal to the source frame size. If the display size is different, the part performing rescale is not related to decoder, so it will be almost impossible to achieve binary match for this, unless you use the same interpolation algorithm.

Memory-Bandwidth Testing of the Decoder

VP9 Random Encoder allows to stress-test decoder memory throughput at inter-prediction. *401_memory_bandwidth.json* parfile contains the encoder settings, which are targeted to maximize the amount of reads of reference frame surfaces. The key distinction of this parfile is usage of 4×4 partitions only. In combination with heavy interpolation utilization and compound prediction, 4×4 block requires reference-frame area of 13×13 pixels to be read twice. Besides, we disable all features using intra-prediction (like intra-only frames or segmentation reference-

frame setting) and ZEROMV mode. To avoid the possibility of single reference prediction mode, we use "--target mem_bw" command-line option, which enforces reference frame sign bias flags to have different values and adds in average 12% reads in inter-prediction.

The stream package contains a sample stream of 1920×1080 resolution. To test other resolutions, you will need to generate test streams using the Random Encoder and the memory-bandwidth parfile.

Worst-Case Performance Testing of the Decoder

In addition to the memory-bandwidth test, we deliver parfiles for decoder-performance stress testing. We provide parfiles *402_worst_performance_19mbps.json*, *403_worst_performance_80mbps.json*, *404_worst_performance_200mbps.json*, which are targeted to stress the most CPU-consuming parts of VP9 decoding pipeline: bool decoder, inter-prediction, and loopfilter.

The main idea is to stress intra-prediction and loopfilter by small-prediction and transform block configuration. The key difference between 19mbps and 80mbps parfiles is the switch from 8×8 prediction blocks to 4×4 ones.

The main tools to control bool-decoder load are QP and mode of forward probability update estimation. 200mbps parfile is derived from the 80mbps one, the only different option is the probability update mode. For the 80mbps case, the encoder estimates the optimal probability based on mode-decision stats, for the 200mbps case it writes fully random probabilities.

Bitrates in the parfile names are approximate and were adjusted to the type of content we use for other streams at Full HD resolution. The Random Encoder doesn't have direct control over bitrate. The main way to adjust the result bitrate are QP-related parameters and the portion of blocks with randomized residual.

Visually Clean Content

Visual parfiles enable you to check decoding results visually when it is impossible to validate the binary match. They are based on the original parfiles, but have some modifications to eliminate visual artifacts: reference-frame scaling, skip-coefficients flag and residual randomization are disabled, range of possible QP is reduced. Random Encoder has a special mode to repeat all invisible frames with show_existing_frame flag. To enable it, add "--target visual" to the encoder options.

Visual parfiles are organized in the same way as standard ones, and have letter "V" in front of the parfile index.

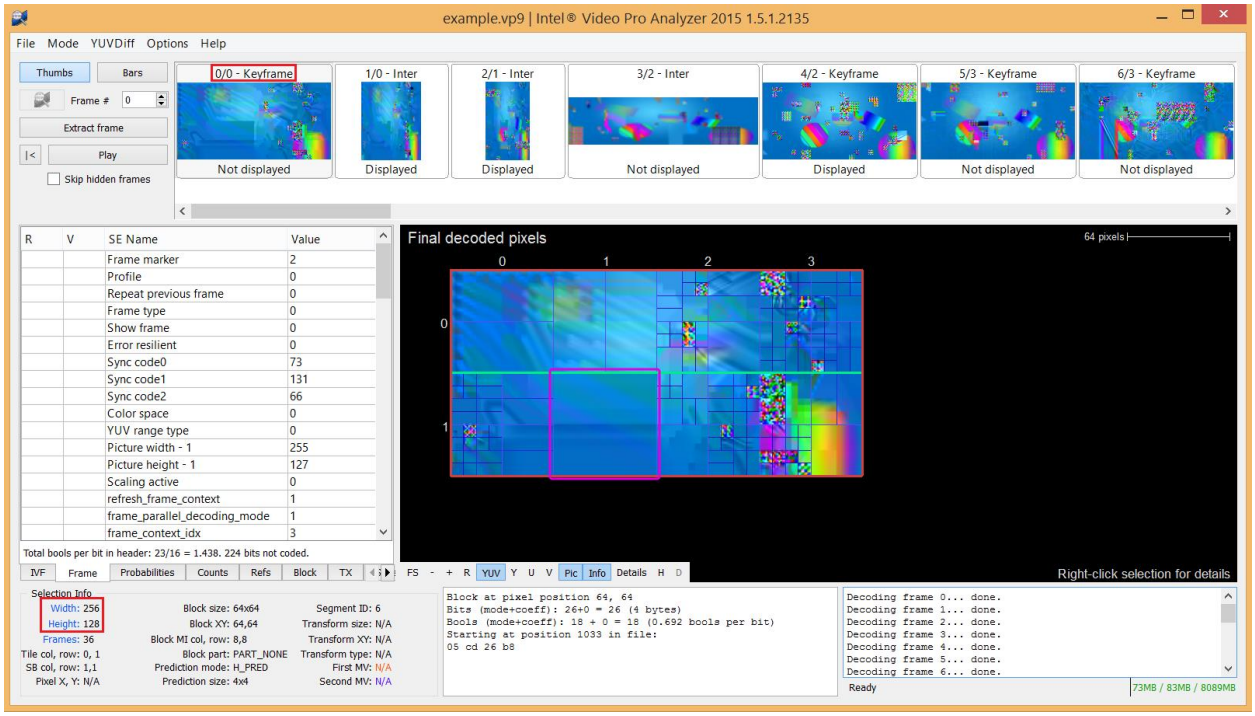
Overriding Parameters

To redefine parameters for specific frames, you should put them into the "override" section of the JSON file. This section has the following structure:

```
"override" : {
    "0,4..6" : { // frames 0, 4, 5 and 6 will use settings specified below instead of the default ones
        "frame" : {
            "frame_type" : [1, 0] // only key frames
        },
        "scaling" : {
            "frame_size_px" : [256, 128] //exact frame size [width, height]
        }
    },
    "1..3" : {
        "frame" : {
            "frame_type" : [0, 1] // only inter frames
        }
    }
}
```

In this example, "0,4..6" is a string specifying the frames to which the new settings will apply. You can set individual frames and frame ranges a..b, separated by commas. Attribute "frame_type" is redefined in both parameter sets shown in this example, making frames 0, 4, 5, 6 key-frames, and 1, 2, 3 - inter-frames. If you do not set any parameters in the “override” parameter set, they are taken from the main parfile.

It’s easy to visualize the effect of these settings using [Intel® Video Pro Analyzer]:



If we open a stream generated with the sample parfile below in Intel® Video Pro Analyzer, we can actually see that frame types are set according to our settings, and frames 0, 4-6 have the size of 256x128 pixels.

Iterating Parameters

Starting from version 2.1, most of the parameter ranges in JSON file can be replaced with iterations:

$$(Start, Step, End)$$

Each call to a certain parameter will return the previous value of this parameter incremented by *Step*, but not greater than *End.* The first used value will be equal to *Start*. When the value exceeds *End*, the iteration starts again from *Start.* This behaviour could be useful if you want to cover a certain range in a determenistic way.

For example, `"frame_type" : [1, 1]` will give key (parameter value 0) and inter (parameter value 1) frames with equal probabilities, while `"frame_type" : (0, 1, 1)` will produce a sequence of alternating frame types (key, inter, key, inter, …).

As another example, `"show_existing_idx"  : (0, 1, 7)` will allow you to cover each possible existing frame buffer index one by one.

Currently, some elements that represent size in the `"scaling"` section are not supported, for example, `"frame_size_px"` or `"min_frame_size"`.

Note that if you specify iterations in brackets, parfile is no longer a valid JSON file.

Error Resilience Encoder

Description

Error Resilience Encoder for VP9 is a tool that allows you to generate broken video streams to test behavior of VP9 decoder on various types of errors. It also allows to control type and positioning of errors that make the stream invalid. This tool can be used to pinpoint flaws in error handling and to define the expected behavior more precisely on a wide range of possible errors.

Error Resilience Encoder is based on Random Encoder: at first, a valid compliant stream is generated with Random Encoder, utilizing all of its flexibility, and then destructive changes are applied to it, based on the user-defined parameters.

As an input, Error Resilience Encoder accepts a *YUV file* and a *parfile* – a JSON-formatted file describing testing settings: features to utilize, fixed values, random values. As an output, Encoder produces an encoded broken bitstream.

Important Restrictions

Although Error Resilience Encoder is an extension of Random Encoder, it's not intended to generate valid streams. Currently, only the first 15 frames of the stream are fully customizable by user (via the "override" section in parfile). For the rest of the frames, if user-defined parameters don't meet the required level of degradation, additional errors are introduced into the stream.

Broken Stream Generation

Rules of error generation are described by the "broken" section in the parfile. If in doubt, see an example of the parfile below, which explains parfile parameters in detail. There are three invalidation options: bitwise randomization, packet-level failures, and corrupted syntax elements. The corresponding subsections in JSON are: "bit", "packet", and "syntax".

Common Parameters

Each of the subsections has string parameter "frames". If this parameter is specified, other parameters of this subsection will only apply to the selected frames. The format of this string is similar to overrides: "0,1..3" means "generate this type of errors only for frames 0, 1, 2 and 3".

If you set the "keep_file_header" parameter in "broken" to a non-zero value, it guards the ivf file header from changes. Otherwise, "ivf_header_prob" probabilities are used to corrupt file header in "bit" and "packet" modes.

Bit Randomization

The idea of bit randomization is to simply invert some bits in a valid VP9 stream. The main parameter here is the "prob" parameter in the "bit" subsection, which defines the probability with which a specific bit will be inverted. Each frame of the generated VP9 stream consists of separately coded parts: container (ivf) header, uncompressed header, compressed header (probabilities), and compressed frame data. Error Resilience Encoders allow to specify separate

probabilities for each of these parts. The corresponding parameters are: "ivf_header_prob", "unc_header_prob", "comp_header_prob", "frame_data_prob". If some of these parameters are not specified explicitly, the default "prob" is used instead. Bit randomization allows to simulate low-level network and storage errors.

Packet Randomization

The main idea is to operate on the packet level, i.e. with the set of bytes. The "prob" in the "packet" section stands for probability for each byte to start the corrupted packet. Parameter "size" : [min, max] determines ranged distribution of the length of this packet. This packet could then be cut from the stream, turned to zeroes, or duplicated, depending on parameter "mode", which is weighted distribution. As a special case, packet-level randomization allows you to cut sections or entire frames from stream. For an example, see parfiles *004_drop_frames.json* and *005_header_mismatch.json* in Error Resilience package.

Syntax Randomization

This section allows you to corrupt syntax elements, where bitstream specification made it possible. It consists mostly of the elements of the uncompressed header. Each parameter is a probability with which this feature will be enabled. The following table lists all these features.

"frame_marker"	Change in constant value, which marks each VP9 frame
"sync_code"	Change in sync code, used in keyframes
"reserved_bit"	Set bit, which is reserved to zero
"header_size"	Corrupt header size, written after uncompressed header
"profile"	Set invalid profile value
"subsampling"	Set 4:2:0 subsampling for profiles 1 and 3
"reference"	Use invalid reference, if possible. Invalid references might be either uninitialized (stream starts with intra-only), or of wrong size (VP9 has restriction on relation between frame sizes), or of wrong profile
"context"	Use uninitialized entropy context
"transform_coeff"	Use coefficients that result in overflow after inverse transform
"tile_count"	Corrupt tile count encoding
"dimensions"	Write wrong dimensions in header

The following values are valid, but not fully covered by randomization in Random Encoder.

"force_write_dimensions"	Explicitly write dimensions on inter-frames
"color_range "	YUV color range
"colorspace"	Random colorspace

Parfiles Description

000_all_features.json	All features written in parfile
001_corrupted_headers.json	Invert bits in headers (ivf, uncompressed, compressed)
002_packet_failures.json	Packet randomization
003_frame_parallel.json	Generated streams contain two independent streams, only one of which is corrupted. Demonstrates use of override feature to create independent streams
004_drop_frames.json	Demonstrates ability to cut entire frames
005_header_mismatch.json	Demonstrates ability to cut specific parts
006_uninitialized_structures.json	Starts with intra-only, randomly generates either uninitialized entropy context, or uninitialized reference
007_invalid_scale_reference.json	Cuts valid reference to produce error
008_missing_existing_frame.json	Cuts first frame, trying to show existing

Parfile Example and Parameter Description

Here we provide a sample parfile, which has randomization of all the features enabled. Each syntax element is provided with a comment about its values. If the type of the parameter (weight or range) is not obvious, check the table below.

```
{
    "frame" : {
        "frame_type"         : [1, 10],         // key | inter
        "start_w_intra_only" : [4, 1],          // no | yes, start bitstream with intra-only instead
                                                //           of key frame
        "tx_mode"            : [1, 1, 1, 1, 2], // 4x4, 8x8, 16x16, 32x32, select
        "interp_filter"      : [1, 1, 1, 1, 4], // reg, smooth, sharp, bilin, switchable
        "horz_tile_range"    : [0, 2],          // log2 val, [0..2]
        "vert_tile_range"    : [0, 6],          // log2 val, [0..6], 2 is max reasonable for FullHD
        "allow_zero_h_tile"  : [0, 1],          // no | yes, allow tiles with zero height
        "frame_context_idx"  : [1, 1, 1, 1],    // context id
        "allow_hp"           : [1, 1],          // 1/8-pel MV accuracy, off/on
        "error_resilient"    : [5, 1],          // no | yes
        "frame_parallel"     : [1, 1],          // no | yes
        "reset_context"      : [1, 1, 1, 1],    // do nothing | invalid | reset current | reset all
        "refresh_context"    : [1, 1],          // no | yes
        // Lossless forces base_qindex and q deltas to zero.
        //     Without this control it would be difficult to generate proper mix of lossless
        //     and lossy frames in one stream.
        "lossless"           : [4, 1],          // no | yes
        "write_display_size" : [1, 1],          // no | yes -- write display size flag
                                                // the frame size is kept from source
        "base_qindex_range"  : [0, 255],        // 0 to 255, zero means lossless
        "y_dc_delta_q_range" : [-15, 15],       // [-15..15]
        "uv_ac_delta_q_range": [-15, 15],       // [-15..15]
        "uv_dc_delta_q_range": [-15, 15],       // [-15..15]
        "lf_filter_level"    : [0, 63],         // [0..63], 6 bit
        "lf_sharpness_level" : [0, 7],          // 3 bit
        "lf_delta_enable"    : [1, 1],          // no | yes
        "lf_delta_update"    : [1, 1],          // no | yes
        "lf_mode_delta_range": [-63, 63],       // 6 bit + sign
        "lf_ref_delta_range" : [-63, 63],       // 6 bit + sign
        "seg_enabled"        : [1, 1],          // no | yes
        "seg_update_map"     : [1, 1],          // no | yes
        "seg_update_data"    : [1, 1],          // no | yes
        "seg_abs_delta"      : [1, 1],          // delta | abs value
        "seg_temporal_update": [1, 1],          // no | yes
        "seg_alt_q_enable"   : [1, 1],          // no | yes
        "seg_alt_q_range"    : [0, 127],        // [0..127],    if seg_abs_delta == 1
        "seg_alt_q_delta"    : [-127, 127],     // [-127..127], if seg_abs_delta == 0
        "seg_alt_lf_enable"  : [1, 1],          // no | yes
        "seg_alt_lf_range"   : [0, 63],         // if seg_abs_delta == 1
        "seg_alt_lf_delta"   : [-63, 63],       // if seg_abs_delta == 0
        "seg_ref_enable"     : [4, 1],          // no | yes
        "seg_ref"            : [1, 1, 1, 1],    // intra | last | golden | altref
        "seg_skip_zeromv"    : [10, 1],         // no | yes

        // Ref-frame-index settings are ignored if not all reference buffers are suitable
        // for prediction: configurations with enabled frame scaling, bit-depth or
        // chroma-subsampling randomization, and when first frame is intra-only and all buffers were
        // initialized.
        "lst_fb_idx_range"   : [0, 7],          // [0..7] last frame ref id range
        "gld_fb_idx_range"   : [0, 7],          // [0..7] golden frame ref id range
        "alt_fb_idx_range"   : [0, 7],          // [0..7] alt frame ref id range
        "refresh_mask_bit"   : [2, 1],          // probs that bit in refresh mask = 0 | 1
        "refresh_mask"       : "01000100",      // explicitly set which frames to refresh;
                                                // overrides refresh_mask_bit
        "ref_frame_sign_bias": [2, 2],          // probs for ref frame sign bias = 0 | 1
        "lst_frame_sign_bias": [1, 0],          // sign bias when last frame is selected as reference;
                                                // when not specified defaults to ref_frame_sign_bias
        "gld_frame_sign_bias": [0, 1],          // sign bias when gold frame is reference
        "alt_frame_sign_bias": [0, 1],          // sign bias when alt frame is reference
        "show_frame"         : [1, 2],          // no | yes
        "intra_only"         : [2, 1],          // no | yes
        "show_existing_frame": [5, 1],          // no | yes - repeat existing frame from ref buffer
        "show_existing_idx"  : [1, 1, 1, 1, 1, 1, 1, 1], // prob weights for existing frame ref idx
```

```
        "comp_pred_mode"     : [1, 1, 1],       // single only | compound only | hybrid
        // frame context update
        //     * no update     -- keep the probabilities from the previous frame
        //     * optimal probs -- estimate optimal probabilities for the frame and write them
        //                        in forward update
        //     * random probs  -- randomize frame context probs for the frame
        "fc_update_type"     : [1, 1, 1],       // no update | optimal probs | random probs
        "fc_rnd_prob_range"  : [1, 255]         // [1..255] (only for random probs)
    },
    "superblock" : {
        "ref_frame"          : [1, 1, 1, 1],    // intra | last | golden | altref
        //                   //DC,  V,  H, 45,135,117,153,207, 63, TM
        "intra_mode"         : [1,  1,  1,  1,  1,  1,  1,  1,  1,  1],
        "inter_mode"         : [1, 1, 1, 1],    // nearestmv | nearmv | zeromv | newmv
        "mv_len"             : 16376,           // max len * 8 (with fractional part bits)
        "skip"               : [9, 1],          // no | yes
        "interp_filter"      : [1, 1, 1],       // reg, smooth, sharp
        "comp_pred"          : [1, 1],          // no | yes, if comp_pred_mode == hybrid
        "comp_pred_var_ref"  : [1, 1],          // ref_idx, 0 or 1
        "seg_id"             : [1, 1, 1, 1, 1, 1, 1, 1], // 8 segments
        "seg_id_predicted"   : [3, 1],          // no | yes

        "partitions"    : [
        //   N, H, V, S
            [1, 1, 1, 1], // 8x8
            [1, 1, 1, 3], // 16x16
            [1, 1, 1, 6], // 32x32
            [1, 1, 1, 9]  // 64x64
        ],

        "tx_size"       : [
        //   4, 8,16,32
            [1, 1],       // blocks less than 16x16
            [1, 1, 1],    // blocks less than 32x32
            [1, 1, 1, 1], // when any transform size suits
        ]
    },
    "color" : {
        // Bitstream profiles indicated by 2-3 bits in the uncompressed header.
        // Profile 0.  8-bit 4:2:0 only.
        // Profile 1.  8-bit 4:4:4, 4:2:2, and 4:4:0.
        // Profile 2.  10-bit and 12-bit color only, with 4:2:0 sampling.
        // Profile 3.  10-bit and 12-bit color only, with 4:2:2/4:4:4/4:4:0 sampling.
        "profile"            : [1, 0, 0, 0], // bitstream profile, 0..3
        "bitdepth"           : [1, 0],       // 10 or 12 bit if profile >= 2
        "subsampling_x"      : [0, 1],       // chroma subsampling, no | yes, if profile == 1 or 3
        "subsampling_y"      : [0, 1]        // chroma subsampling, no | yes, if profile == 1 or 3
    },
    "scaling" : {
        // Input file defines maximum resolution.
        // intra_max_downscale pair defines maximum possible downscale on key frames and intra-only
        // frames for width and height, respectively.
        //
        //    For example, if we have source 1920x1080 and "intra_max_downscale" : [2, 4],
        //    then frame width for intra frames will vary in range of 960..1920 (up to 2x downscale)
        //    and the range for frame height will be 270..1080 (up to 4x downscale).
        //
        // To randomize resolution of inter frames, we do the following:
        // 1. Pick one reference frame with ref frame index from range defined by "base_ref_frame"
        //    parameter.
        // 2. For each dimension, decide whether it will be increased or decresed by chances from
        //    "downscale_or_upscale".
        // 3. Pick random value in range between the original size and the size defined by max
        //    upscale/downscale factor.
        "resolution_change"    : [0, 1],   // probs of resolution change <no | yes>
        "scale_first_frame"    : "yes",    // "no" | "maybe" | "yes"
        "intra_max_downscale"  : [16, 16], // width, height (floating point allowed)
        "base_ref_frame"       : [0, 7],   // ref frame to borrow resolution if size_from_ref == 1
        "downscale_or_upscale" : [1, 1],   // chances to decrease and increase frame size
        "max_upscale_factor"   : [16, 16], // width, height (floating point allowed), max = 16
        "max_downscale_factor" : [2, 2],   // width, height (floating point allowed), max = 2
        "min_frame_size"       : [4, 4]    // width and height in pixels,
                                           // min frame size restrictions
    },
```

```
    "residual" : {
        // Only one mode of residual randomization per stream can be used.
        "type"           : "diff",      // off | diff | transform
        "sb_randomized"  : [9, 1],      // no | yes
        "residual_range" : [-255, 255], // min and max residual value (max range is [-255, 255])

        //ZERO, ONE, TWO, THREE, FOUR, CAT1, CAT2, CAT3, CAT4, CAT5, CAT6
        "coef_tokens" : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

        // For tx mode if inverse transform of random-generated block doesn't produce valid
        // residual, encoder tries to regerate it, and stops after several unsuccessful tries.
        // Increasing the range will reduce the chances to generate valid transform and the portion
        // of random blocks in bitstream.
        "tx_eob_range" : [
            [1, 16],   // tx 4x4
            [1, 64],   // tx 8x8
            [1, 256],  // tx 16x16
            [1, 1024]  // tx 32x32
        ]
    },
    "broken" : {
        "keep_file_header" : 1,
        "bit" : {
            "frames" : "0..5,10..15",
            //"prob" : 0.005,

            "ivf_header_prob" : 0.001,
            "unc_header_prob" : 0.001,
            "comp_header_prob" : 0.0001,
            "frame_data_prob" : 0.00001
        },
        "packet" : {
            "frames" : "6,8,10,12",
            "size" : [2, 8],
            //"prob" : 0.001,

            "ivf_header_prob" : 0,
            "unc_header_prob" : 0.01,
            "comp_header_prob" : 0.001,
            "frame_data_prob" : 0.0001,
            "mode" : [1, 1, 1] //loss, duplicate, zero
        },
        "syntax" : {
            "frames" : "7..9,11,13",
            //"prob" : 0.1,

            "frame_marker" : 0.2,
            "sync_code" : 0.2,
            "reserved_bit" : 0.2,
            "header_size" : 0.2,
            "profile" : 0.2,
            "subsampling" : 0,
            "reference" : 0.5,
            "context" : 0.4,
            "color_range" : 0,
            "colorspace" : 0,
            "dimensions" : 0,
            "force_write_dimensions" : 1,
            "tile_count" : 0,
            "transform_coeff" : 1,
        }
    },


    "override" : {
        "0,4..6" : { // frames 0, 4, 5 and 6 will use settings specified below instead of default ones
            "frame" : {
                "frame_type" : [1, 0] // only key frames
            },
            "scaling" : {
                "frame_size_px" : [256, 128] //exact frame size [width, height]
            }
```

```
        },
        "1..3" : {
            "frame" : {
                "frame_type" : [0, 1] // only inter frames
            }
        }
    }
}
```

Table 1. Randomization-control parameter types and acceptable values

Parameter	Type	Enumeration (for weights) and acceptable range (for other types)	Default value
Frame section			
frame_type	weights	KEY, INTER	[1, 24]
start_w_intra_only	weights	False, True	[1, 0]
tx_mode	weights	ONLY_4X4, ALLOW_8X8, ALLOW_16X16, ALLOW_32X32, TX_MODE_SELECT	[1, 1, 1, 1, 1]
interp_filter	weights	EIGHTTAP, EIGHTTAP_SMOOTH, EIGHTTAP_SHARP, BILINEAR, SWITCHABLE	[1, 1, 1, 1, 1]
horz_tile_range	range	[0..2]	[0, 0]
vert_tile_range	range	[0..6]	[0, 0]
allow_zero_h_tile	weights	False, True	[0, 1]
frame_context_idx	weights	0, 1, 2, 3	[1, 1, 1, 1]
allow_hp	weights	False, True	[1, 1]
error_resilient	weights	False, True	[1, 0]
frame_parallel	weights	False, True	[1, 0]
reset_context	weights	0, 1, 2, 3	[1, 0]
refresh_context	weights	False, True	[1, 1]
lossless	weights	False, True	[1, 0]
write_display_size	weights	False, True	[1, 1]
base_qindex_range	range	[0..255]	[40, 40]
y_dc_delta_q_range	range	[−15..15]	[0, 0]
uv_ac_delta_q_range	range	[−15..15]	[0, 0]
uv_dc_delta_q_range	range	[−15..15]	[0, 0]
lf_filter_level	range	[0..63]	[0, 63]
lf_sharpness_level	range	[0..7]	[0, 7]
lf_delta_enable	weights	False, True	[1, 1]
lf_delta_update	weights	False, True	[1, 1]
lf_mode_delta_range	range	[−63, 63]	[−63, 63]
lf_ref_delta_range	range	[−63, 63]	[−63, 63]
seg_enabled	weights	False, True	[1, 1]
seg_update_map	weights	False, True	[1, 1]
seg_update_data	weights	False, True	[1, 1]
seg_abs_delta	weights	False, True	[1, 1]
seg_temporal_update	weights	False, True	[1, 1]
seg_alt_q_enable	weights	False, True	[1, 1]
seg_alt_q_range	range	[0, 127]	[20, 40]
seg_alt_q_delta	range	[−127, 127]	[−20, 20]
seg_alt_lf_enable	weights	False, True	[1, 1]
seg_alt_lf_range	range	[0, 63]	[0, 63]
seg_alt_lf_delta	range	[-63, 63]	[-63, 63]
seg_ref_enable	weights	False, True	[1, 1]
seg_ref	weights	INTRA_FRAME, LAST_FRAME, GOLDEN_FRAME, ALTREF_FRAME	[1, 1, 1, 1]
seg_skip_zeromv	weights	False, True	[1, 1]
lst_fb_idx_range	range	[0..7]	[0, 0]
gld_fb_idx_range	range	[0..7]	[1, 1]
alt_fb_idx_range	range	[0..7]	[2, 2]
refresh_mask_bit	weights	0, 1	[1, 2]
refresh_mask	bitmask	[00000000..11111111]	refresh_mask_bit
ref_frame_sign_bias	weights	0, 1	[2, 1]

lst_frame_sign_bias	weights	0, 1	ref_frame_sign_bias
alt_frame_sign_bias	weights	0, 1	ref_frame_sign_bias
gld_frame_sign_bias	weights	0, 1	ref_frame_sign_bias
show_frame	weights	False, True	[0, 1]
intra_only	weights	False, True	[2, 1]
show_existing_frame	weights	False, True	[1, 0]
show_existing_idx	weights	0, 1, 2, 3, 4, 5, 6, 7	[1, 1, 1, 1, 1, 1, 1, 1]
comp_pred_mode	weights	SINGLE_REFERENCE, COMPOUND_REFERENCE, REFERENCE_MODE_SELECT	[2, 1, 2]
fc_update_type	weights	No update, optimal probabilities, random probabilities	[1, 1, 1]
fc_rnd_prob_range	range	[1..255]	[1, 254]
Superblock section			
ref_frame	weights	INTRA_FRAME, LAST_FRAME, GOLDEN_FRAME, ALTREF_FRAME	[1, 1, 1, 1]
intra_mode	weights	DC_PRED, V_PRED, H_PRED, D45_PRED, D135_PRED, D117_PRED, D153_PRED, D207_PRED, D63_PRED, TM_PRED	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
inter_mode	weights	NEARESTMV, NEARMV, ZEROMV, NEWMV	[1, 1, 1, 1]
mv_len	integer	0..16384	1280
mv_joint_zero	weights	False, True	[1, 0]
skip	weights	False, True	[10, 1]
interp_filter	weights	EIGHTTAP, EIGHTTAP_SMOOTH, EIGHTTAP_SHARP	[1, 1, 1]
comp_pred	weights	SINGLE_REFERENCE, COMPOUND_REFERENCE	[1, 1]
comp_pred_var_ref	weights	0, 1	[1, 1]
seg_id	weights	0, 1, 2, 3, 4, 5, 6, 7	[1, 1, 1, 1, 1, 1, 1, 1]
seg_id_predicted	weights	False, True	[3, 1]
partitions	weights	PARTITION_NONE, PARTITION_HORZ, PARTITION_VERT, PARTITION_SPLIT	// N, H, V, S [1, 1, 1, 1], // 8x8 [1, 1, 1, 3], // 16x16 [1, 1, 1, 6], // 32x32 [1, 1, 1, 9] // 64x64
tx_size	weights	TX_4X4, TX_8X8, TX_16X16, TX_32X32	[1, 1], [1, 1, 1], [1, 1, 1, 1]
Color section			
profile	weights	Profile 0, profile 1, profile 2, profile 3	[1, 0, 0, 0]
bit_depth	weights	10 bit, 12 bit	[1, 0]
subsampling_x	weights	False, True	[0, 1]
subsampling_y	weights	False, True	[0, 1]
Scaling section			
resolution_change	weights	False, True	[1, 0]
frame_size_px	int	[width, height], no more than input_frame_size	[width, height]
intra_max_downscale	float	[1.0 .. 16.0]	1.0
base_ref_frame [0, 7],	range	[0..7]	[0, 7]
downscale_or_upscale	weights	Downscale, upscale	[1, 1]
max_upscale_factor	float	[1.0 .. 16.0]	1.0
max_downscale_factor	float	[1.0 .. 2.0]	1.0
scale_first_frame	string	"no", "maybe", "yes"	"no"
min_frame_size	Int	[1..input_frame_size]	[4, 4]
Residual section			
type	string	"off", "diff", "transform", "tokens"	"off"
sb_randomized	weights	False, True	[1, 0]
residual_range	range	[−255..255]	[−255, 255]
coef_tokens	weights	ZERO, ONE, TWO, THREE, FOUR, CAT1-CAT6	Equal probability
tx_eob_range	range	[0, 16], // tx 4x4 [0, 64], // tx 8x8 [0, 256], // tx 16x16 [0, 1024] // tx 32x32	[0, 16], // tx 4x4 [0, 64], // tx 8x8 [0, 256], // tx 16x16 [0, 1024] // tx 32x32

Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting **www.intel.com/design/literature.htm**.

Intel, the Intel logo, Intel Core are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804