The triangle tree record contains the structure of a triangle spanning tree which links all the triangles of the corresponding connected component forming a simple polygon. The 3D mesh is represented in a triangulated form in the bitstream, which also contains the information necessary to reconstruct the original faces. The vertex graph record contains the information necessary to stitch pairs of boundary edges of the simple polygon to reconstruct the original connectivity, not only within the current connected component, but also to previously decoded connected components. The connectivity information is categorized as *global* information (per connected component) and *local* information (per triangle). The global information is stored in the Vertex Graph and Triangle Tree records. The local information is stored in the Triangle Data record. The triangle data is arranged on a per triangle basis, where the ordering of the triangles is determined by the traversal of the triangle tree.

Data for triangle #1	Data for triangle #2	...	Data for triangle #nT

The data for a given triangle is organized as follows:

marching edge	td_orientation	polygon_edge	coord	normal	color	texCoord

The *marching edge*, *td_orientation* and *polygon_edge* constitute the per triangle connectivity information. The other fields contain information to reconstruct the vertex coordinates (*coord*) and optionally, normal, color, and texture coordinate (*texCoord*) information.

If the 3D mesh is encoded in hierarchical mode, each Forest Split data block is composed of an optional pre-smoothing data block, an optional post-smoothing data block, a pre-update data block, an optional smoothing constraints data block, a post-update data block, and an optional other-update data block.

Pre-smoothing	Post-smoothing	Pre-update	Constraints	Post-update	Other-update

The pre-smoothing data block contains the parameters used to apply a smoothing step as a global predictor for vertex coordinates. The post-smoothing data block contains the parameters used to apply a smoothing step to the vertex coordinates to remove quantisation artifacts. The pre update data block contains the information necessary to update the connectivity, and the property data updates for properties bound per-face and per-corner to the new faces created by the connectivity update. The smoothing constraints data block contains information used to perform the smoothing steps with sharp edge discontinuities and fixed vertices. After the connectivity update is applied, the pre-smoothing operation specified by the parameters stored in the pre-smoothing data block is applied as a global predictor for the vertex coordinates. The post update data block contains tree loop vertex coordinate updates, with respect to the vertex coordinates predicted by the pre-smoothing step, if applied; normal, color, and texture coordinate updates for properties bound per-vertex to tree loop vertices; normal, and color updates for properties bound per-face to tree loop faces; and normal, color, and texture coordinate updates for properties bound per-corner to tree loop corners. The other-update data block contains vertex coordinate updates and property updates for all the vertices, faces, and corners not updated by data included in the post-update data block.

6.2 Visual bitstream syntax

6.2.1 Start codes

Start codes are specific bit patterns that do not otherwise occur in the video stream.

Each start code consists of a start code prefix followed by a start code value. The start code prefix is a string of twenty three bits with the value zero followed by a single bit with the value one. The start code prefix is thus the bit string ‘ 0000 0000 0000 0000 0000 0001’.

The start code value is an eight bit integer which identifies the type of start code. Many types of start code have just one start code value. However video_object_start_code and video_object_layer_start_code are represented by many start code values.

All start codes shall be byte aligned. This shall be achieved by first inserting a bit with the value zero and then, if necessary, inserting bits with the value one before the start code prefix such that the first bit of the start code prefix is the first (most significant) bit of a byte. For stuffing of 1 to 8 bits, the codewords are as follows in Table 6-2.

Table 6-2-- Stuffing codewords

Bits to be stuffed	Stuffing Codeword
1	0
2	01
3	011
4	0111
5	01111
6	011111
7	0111111
8	01111111

Table 6-3 defines the start code values for all start codes used in the visual bitstream.

Table 6-3 — Start code values

name	start code value (hexadecimal)
video_object_start_code	00 through 1F
video_object_layer_start_code	20 through 2F
reserved	30 through AF
visual_object_sequence_start_code	B0
visual_object_sequence_end_code	B1
user_data_start_code	B2
group_of_vop_start_code	B3
video_session_error_code	B4
visual_object_start_code	B5
vop_start_code	B6
reserved	B7-B9
fba_object_start_code	BA
fba_object_plane_start_code	BB
mesh_object_start_code	BC
mesh_object_plane_start_code	BD
still_texture_object_start_code	BE
texture_spatial_layer_start_code	BF
texture_snr_layer_start_code	C0
texture_tile_start_code	C1
texture_shape_layer_start_code	C2
reserved	C3-C5
System start codes (see note)	C6 through FF
NOTE System start codes are defined in ISO/IEC 14496-1	

The use of the start codes is defined in the following syntax description with the exception of the video_session_error_code. The video_session_error_code has been allocated for use by a media interface to indicate where uncorrectable errors have been detected.

This syntax for visual bitstreams defines two types of information:

- 1. Configuration information
- a. Global configuration information, referring to the whole group of visual objects that will be simultaneously decoded and composited by a decoder (VisualObjectSequence()).
- b. Object configuration information, referring to a single visual object (VO). This is associated with VisualObject().
- c.Object layer configuration information, referring to a single layer of a single visual object (VOL) VisualObjectLayer()
- 2. Elementary stream data, containing the data for a single layer of a visual object.

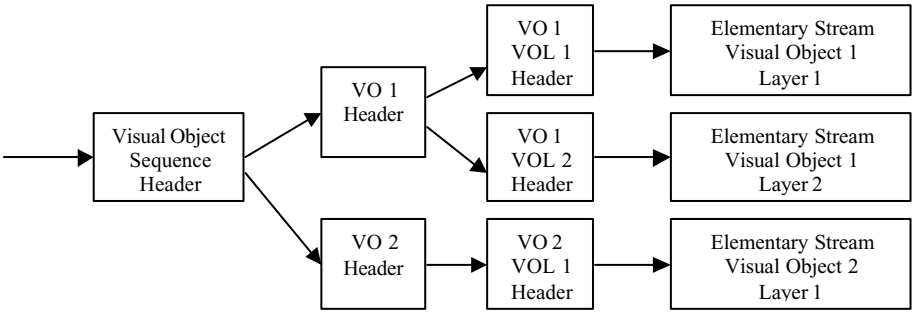

Figure 6-11 -- Example Visual Information – Logical Structure

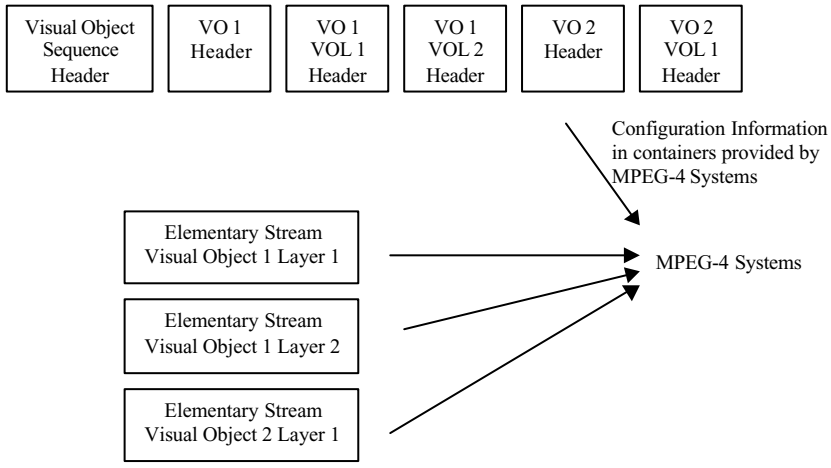

Figure 6-12 -- Example Visual Bitstream – Separate Configuration Information / Elementary Stream.

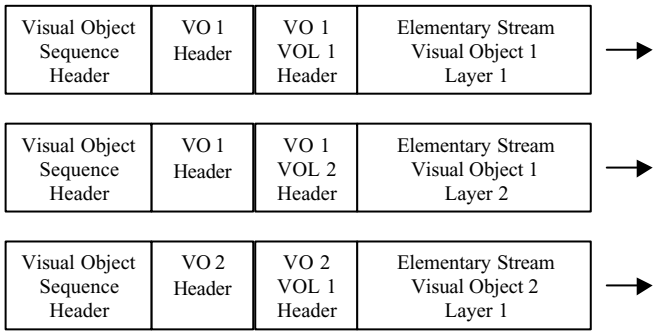

Figure 6-13 -- Example Visual Bitstream – Combined Configuration Information / Elementary Stream

The following functions are entry points for elementary streams, and entry into these functions defines the breakpoint between configuration information and elementary streams:

- 1. Group_of_VideoObjectPlane(),
- 2. VideoObjectPlane(),
- 3. video_plane_with_short_header(),
- 4. MeshObject(),
- 5. fba_object().

For still texture objects, configuration information ends and elementary stream data begins in StilTextureObject() immediately before the first call to wavelet_dc_decode(), as indicated by the comment in subclause 6.2.8.

There is no overlap of syntax between configuration information and elementary streams.

The configuration information contains all data that is not part of an elementary stream, including that defined by VisualObjectSequence(), VisualObject() and VideoObjectLayer().

ISO/IEC 14496-2 does not provide for the multiplexing of multiple elementary streams into a single bitstream. One visual bitstream contains exactly one elementary stream, which describes one layer of one visual object. A visual decoder must conceptually have a separate entry port for each layer of each object to be decoded.

Visual objects coded in accordance with this Part may be carried within a Systems bitstream as defined by ISO/IEC 14496-1. The coded visual objects may also be free standing or carried within other types of systems. Configuration information may be carried separately from or combined with elementary stream data:

1. *Separate Configuration / Elementary Streams (e.g. Inside ISO/IEC 14496-1 Bitstreams)*

When coded visual objects are carried within a Systems bitstream defined by ISO/IEC 14496-1, configuration information and elementary stream data are always carried separately. Configuration information and elementary streams follow the syntax below, subject to the break points between them defined above. The Systems specification ISO/IEC 14496-1 defines containers that are used to carry Visual Object Sequence, Visual Object and Video Object Layer configuration information. For video objects one container is used for each layer for each object. This container carries a Visual Object Sequence header, a Visual Object header and a Video Object Layer header. For other types of visual objects, one container per visual object is used. This container carries a Visual Object Sequence header and a Visual Object header. The Visual Object Sequence Header must be identical for all

visual streams input simultaneously to a decoder. The Visual Object Headers for each layer of a multilayer object must be identical.

2. *Combined Configuration / Elementary Streams*

The elementary stream data associated with a single layer may be wrapped in configuration information defined in accordance with the syntax below. A visual bitstream may contain at most one instance of each of VisualObjectSequence(), VisualObject() and VideoObjectLayer(), with the exception of repetition of the Visual Object Sequence Header, the Visual Object Header and the Video Object Layer Header as described below. The Visual Object Sequence Header must be identical for all visual streams input simultaneously to a decoder. The Visual Object Headers for each layer of a multilayer object must be identical.

The Visual Object Sequence Header, the Visual Object Header and the Video Object Layer Header may be repeated in a single visual bitstream. Repeating these headers enables random access into the visual bitstream and recovery of these headers when the original headers are corrupted by errors. This header repetition is used only when visual_object_type in the Visual Object Header indicates that visual object type is video. (i.e. visual_object_type=="video ID") All of the data elements in the Visual Object Sequence Header, the Visual Object Header and the Video Object Layer Header repeated in a visual bitstream shall have the same value as in the original headers, except that first_half_vbv_occupancy and latter_half_vbv_occupancy may be changed to specify the VBV occupancy just before the removal of the first VOP following the repeated Video Object Layer Header.

6.2.2 Visual Object Sequence and Visual Object

VisualObjectSequence() {	No. of bits	Mnemonic
do {		
visual_object_sequence_start_code	32	bslbf
profile_and_level_indication	8	uimsbf
while (next_bits()== user_data_start_code){		
user_data()		
}		
VisualObject()		
} while (next_bits() != visual_object_sequence_end_code)		
visual_object_sequence_end_code	32	bslbf
}		

VisualObject() {	**No. of bits**	**Mnemonic**
visual_object_start_code	32	bslbf
is_visual_object_identifier	1	uimsbf
if (is_visual_object_identifier) {		
visual_object_verid	4	uimsbf
visual_object_priority	3	uimsbf
}		
visual_object_type	4	uimsbf
if (visual_object_type == "video ID" || visual_object_type == " still texture ID") {		
video_signal_type()		
}		
next_start_code()		
while (next_bits()== user_data_start_code){		
user_data()		
}		
if (visual_object_type == "video ID") {		

video_object_start_code	32	bslbf
VideoObjectLayer()		
}		
else if (visual_object_type == " still texture ID") {		
StillTextureObject()		
}		
else if (visual_object_type == "mesh ID") {		
MeshObject()		
}		
else if (visual_object_type == "FBA ID") {		
FBAObject()		
}		
else if (visual_object_type == " 3D mesh ID") {		
3D_Mesh_Object()		
}		
if (next_bits() != "0000 0000 0000 0000 0000 0001")		
next_start_code()		
}		

video_signal_type() {	**No. of bits**	**Mnemonic**
video_signal_type	1	bslbf
if (video_signal_type) {		
video_format	3	uimsbf
video_range	1	bslbf
colour_description	1	bslbf
if (colour_description) {		
colour_primaries	8	uimsbf
transfer_characteristics	8	uimsbf
matrix_coefficients	8	uimsbf
}		
}		
}		

6.2.2.1 User data

user_data() {	**No. of bits**	**Mnemonic**
user_data_start_code	32	bslbf
while(next_bits() != '0000 0000 0000 0000 0000 0001') {		
user_data	8	uimsbf
}		
}		

6.2.3 Video Object Layer

VideoObjectLayer() {	No. of bits	Mnemonic
if(next_bits() == video_object_layer_start_code) {		
short_video_header = 0		
video_object_layer_start_code	32	bslbf
random_accessible_vol	1	bslbf
video_object_type_indication	8	uimsbf
is_object_layer_identifier	1	uimsbf
if (is_object_layer_identifier) {		
video_object_layer_verid	4	uimsbf
video_object_layer_priority	3	uimsbf
}		
aspect_ratio_info	4	uimsbf
if (aspect_ratio_info == “extended_PAR”) {		
par_width	8	uimsbf
par_height	8	uimsbf
}		
vol_control_parameters	1	bslbf
if (vol_control_parameters) {		
chroma_format	2	uimsbf
low_delay	1	uimsbf
vbv_parameters	1	blsbf
if (vbv_parameters) {		
first_half_bit_rate	15	uimsbf
marker_bit	1	bslbf
latter_half_bit_rate	15	uimsbf
marker_bit	1	bslbf
first_half_vbv_buffer_size	15	uimsbf
marker_bit	1	bslbf
latter_half_vbv_buffer_size	3	uimsbf
first_half_vbv_occupancy	11	uimsbf
marker_bit	1	blsbf
latter_half_vbv_occupancy	15	uimsbf
marker_bit	1	blsbf
}		
}		
video_object_layer_shape	2	uimsbf
if (video_object_layer_shape == "grayscale" && video_object_layer_verid != ‘0001’)		
video_object_layer_shape_extension	4	uimsbf
marker_bit	1	bslbf
vop_time_increment_resolution	16	uimsbf
marker_bit	1	bslbf
fixed_vop_rate	1	bslbf
if (fixed_vop_rate)		
fixed_vop_time_increment	1-16	uimsbf
if (video_object_layer_shape != “binary only”) {		
if (video_object_layer_shape == “ rectangular”) {		

marker_bit	1	bslbf
video_object_layer_width	13	uimsbf
marker_bit	1	bslbf
video_object_layer_height	13	uimsbf
marker_bit	1	bslbf
}		
interlaced	1	bslbf
obmc_disable	1	bslbf
if (video_object_layer_verid == '0001')		
sprite_enable	1	bslbf
else		
sprite_enable	2	uimsbf
if (sprite_enable == " static" || sprite_enable == "GMC") {		
if (sprite_enable != "GMC") {		
sprite_width	13	uimsbf
marker_bit	1	bslbf
sprite_height	13	uimsbf
marker_bit	1	bslbf
sprite_left_coordinate	13	simsbf
marker_bit	1	bslbf
sprite_top_coordinate	13	simsbf
marker_bit	1	bslbf
}		
no_of_sprite_warping_points	6	uimsbf
sprite_warping_accuracy	2	uimsbf
sprite_brightness_change	1	bslbf
if (sprite_enable != "GMC")		
low_latency_sprite_enable	1	bslbf
}		
if (video_object_layer_verid != '0001' && video_object_layer_shape != "rectangular")		
sadct_disable	1	bslbf
not_8_bit	1	bslbf
if (not_8_ bit) {		
quant_precision	4	uimsbf
bits_per_pixel	4	uimsbf
}		
if (video_object_layer_shape=="grayscale") {		
no_gray_quant_update	1	bslbf
composition_method	1	bslbf
linear_composition	1	bslbf
}		
quant_type	1	bslbf
if (quant_type) {		
load_intra_quant_mat	1	bslbf
if (load_intra_quant_mat)		
intra_quant_mat	8*[2-64]	uimsbf

load_nonintra_quant_mat	1	bslbf
if (load_nonintra_quant_mat)		
nonintra_quant_mat	8*[2-64]	uimsbf
if(video_object_layer_shape=="grayscale") {		
for(i=0; i<aux_comp_count; i++) {		
load_intra_quant_mat_grayscale	1	bslbf
if(load_intra_quant_mat_grayscale)		
intra_quant_mat_grayscale[i]	8*[2-64]	uimsbf
load_nonintra_quant_mat_grayscale	1	bslbf
if(load_nonintra_quant_mat_grayscale)		
nonintra_quant_mat_grayscale [i]	8*[2-64]	uimsbf
}		
}		
}		
if (video_object_layer_verid != '0001')		
quarter_sample	1	bslbf
complexity_estimation_disable	1	bslbf
if (!complexity_estimation_disable)		
define_vop_complexity_estimation_header()		
resync_marker_disable	1	bslbf
data_partitioned	1	bslbf
if(data_partitioned)		
reversible_vlc	1	bslbf
if(video_object_layer_verid != '0001') {		
newpred_enable	1	bslbf
if (newpred_enable) {		
requested_upstream_message_type	2	uimsbf
newpred_segment_type	1	bslbf
}		
reduced_resolution_vop_enable	1	bslbf
}		
scalability	1	bslbf
if (scalability) {		
hierarchy_type	1	bslbf
ref_layer_id	4	uimsbf
ref_layer_sampling_direc	1	bslbf
hor_sampling_factor_n	5	uimsbf
hor_sampling_factor_m	5	uimsbf
vert_sampling_factor_n	5	uimsbf
vert_sampling_factor_m	5	uimsbf
enhancement_type	1	bslbf

if(video_object_layer == " binary" && hierarchy_type== '0') {		
use_ref_shape	1	bslbf
use_ref_texture	1	bslbf
shape_hor_sampling_factor_n	5	uimsbf
shape_hor_sampling_factor_m	5	uimsbf
shape_vert_sampling_factor_n	5	uimsbf
shape_vert_sampling_factor_m	5	uimsbf
}		
}		
}		
else {		
if(video_object_layer_verid !="0001") {		
scalability	1	bslbf
if(scalability) {		
shape_hor_sampling_factor_n	5	uimsbf
shape_hor_sampling_factor_m	5	uimsbf
shape_vert_sampling_factor_n	5	uimsbf
shape_vert_sampling_factor_m	5	uimsbf
}		
}		
resync_marker_disable	1	bslbf
}		
next_start_code()		
while (next_bits()== user_data_start_code){		
user_data()		
}		
if (sprite_enable == "static" && !low_latency_sprite_enable)		
VideoObjectPlane()		
do {		
if (next_bits() == group_of_vop_start_code)		
Group_of_VideoObjectPlane()		
VideoObjectPlane()		
} while ((next_bits() == group_of_vop_start_code) || (next_bits() == vop_start_code))		
} else {		
short_video_header = 1		
do {		
video_plane_with_short_header()		
} while(next_bits() == short_video_start_marker)		
}		
}		

define_vop_complexity_estimation_header() {	**No. of bits**	**Mnemonic**
estimation_method	2	uimsbf

if (estimation_method ==’00’ || estimation_method == ‘01’) {		
shape_complexity_estimation_ disable	1	
if (!shape_complexity_estimation_disable) {		bslbf
opaque	1	bslbf
transparent	1	bslbf
intra_cae	1	bslbf
inter_cae	1	bslbf
no_update	1	bslbf
upsampling	1	bslbf
}		
texture_complexity_estimation_set_1_disable	1	bslbf
if (!texture_complexity_estimation_set_1_disable) {		
intra_blocks	1	bslbf
inter_blocks	1	bslbf
inter4v_blocks	1	bslbf
not_coded_blocks	1	bslbf
}		
marker_bit	1	bslbf
texture_complexity_estimation_set_2_disable	1	bslbf
if (!texture_complexity_ estimation_set_2_disable) {		
dct_coefs	1	bslbf
dct_lines	1	bslbf
vlc_symbols	1	bslbf
vlc_bits	1	bslbf
}		
motion_compensation_complexity_disable	1	bslbf
If (!motion_compensation_complexity_disable) {		
apm	1	bslbf
npm	1	bslbf
interpolate_mc_q	1	bslbf
forw_back_mc_q	1	bslbf
halfpel2	1	bslbf
halfpel4	1	bslbf
}		
marker_bit	1	bslbf
if(estimation_method == ‘01’) {		
version2_complexity_estimation_disable	1	bslbf
if (!version2_complexity_estimation_disable) {		
sadct	1	bslbf
quarterpel	1	bslbf
}		
}		
}		
}		

6.2.4 Group of Video Object Plane

Group_of_VideoObjectPlane() {	No. of bits	Mnemonic
group_of_vop_start_code	32	bslbf
time_code	18	
closed_gov	1	bslbf
broken_link	1	bslbf
next_start_code()		
while (next_bits()== user_data_start_code){		
user_data()		
}		
}		

6.2.5 Video Object Plane and Video Plane with Short Header

VideoObjectPlane() {	No. of bits	Mnemonic
vop_start_code	32	bslbf
vop_coding_type	2	uimsbf
do {		
modulo_time_base	1	bslbf
} while (modulo_time_base != '0')		
marker_bit	1	bslbf
vop_time_increment	1-16	uimsbf
marker_bit	1	bslbf
vop_coded	1	bslbf
if (vop_coded == '0') {		
next_start_code()		
return()		
}		
if (newpred_enable) {		
vop_id	4-15	uimsbf
vop_id_for_prediction_indication	1	bslbf
if (vop_id_for_prediction_indication)		
vop_id_for_prediction	4-15	uimsbf
marker_bit	1	bslbf
}		
if ((video_object_layer_shape != " binary only") && (vop_coding_type == "P" || (vop_coding_type == "S" && sprite_enable == " GMC")))		
vop_rounding_type	1	bslbf
if ((reduced_resolution_vop_enable) && (video_object_layer_shape == "rectangular") && ((vop_coding_type == "P") || (vop_coding_type == " I")))		
vop_reduced_resolution	1	bslbf
if (video_object_layer_shape != "rectangular") {		
if(!(sprite_enable == "static" && vop_coding_type == "I")) {		

vop_width	13	uimsbf
marker_bit	1	bslbf
vop_height	13	uimsbf
marker_bit	1	bslbf
vop_horizontal_mc_spatial_ref	13	simsbf
marker_bit	1	bslbf
vop_vertical_mc_spatial_ref	13	simsbf
marker_bit	1	bslbf
}		
if ((video_object_layer_shape != " binary only") && scalability && enhancement_type)		
background_composition	1	bslbf
change_conv_ratio_disable	1	bslbf
vop_constant_alpha	1	bslbf
if (vop_constant_alpha)		
vop_constant_alpha_value	8	bslbf
}		
if (video_object_layer_shape != "binary only")		
if (!complexity_estimation_disable)		
read_vop_complexity_estimation_header()		
if (video_object_layer_shape != "binary only") {		
intra_dc_vlc_thr	3	uimsbf
if (interlaced) {		
top_field_first	1	bslbf
alternate_vertical_scan_flag	1	bslbf
}		
}		
if ((sprite_enable ==" static" || sprite_enable=="GMC") && vop_coding_type == "S") {		
if (no_of_sprite_warping_points > 0)		
sprite_trajectory()		
if (sprite_brightness_change)		
brightness_change_factor()		
if(sprite_enable == "static") {		
if (sprite_transmit_mode != " stop" && low_latency_sprite_enable) {		
do {		
sprite_transmit_mode	2	uimsbf
if ((sprite_transmit_mode == "piece") || (sprite_transmit_mode == "update"))		
decode_sprite_piece()		
} while (sprite_transmit_mode != "stop" && sprite_transmit_mode != "pause")		
}		
next_start_code()		
return()		
}		
}		
if (video_object_layer_shape != "binary only") {		

vop_quant	3-9	uimsbf
if(video_object_layer_shape==”grayscale”)		
for(i=0; i<aux_comp_count; i++)		
vop_alpha_quant[i]	6	uimsbf
if (vop_coding_type != “ I”)		
vop_fcode_forward	3	uimsbf
if (vop_coding_type == “B”)		
vop_fcode_backward	3	uimsbf
if (!scalability) {		

if (video_object_layer_shape != “rectangular” && vop_coding_type != “I”)
vop_shape_coding_type
motion_shape_texture()
while (nextbits_bytealigned() == resync_marker) {
	1	bslbf

video_packet_header()		
motion_shape_texture()		
}		
}		
else {		
if (enhancement_type) {		
load_backward_shape	1	bslbf
if (load_backward_shape) {		
backward_shape_width	13	uimsbf
marker_bit	1	bslbf
backward_shape_ height	13	uimsbf
marker_bit	1	bslbf
backward_shape_horizontal_mc_spatial_ref	13	simsbf
marker_bit	1	bslbf
backward_shape_vertical_mc_spatial_ref	13	simsbf
backward_shape()		
load_forward_shape	1	bslbf
if (load_forward_shape) {		
forward_shape_width **marker_bit** **forward_shape_height** **marker_bit** **forward_shape_horizontal_mc_spatial_ref** **marker_bit** **forward_shape_vertical_mc_spatial_ref**	13 1 13 1 13 1 13	uimsbf bslbf uimsbf bslbf simsbf bslbf simsbf
forward_shape()		
}		
}		
}		
ref_select_code	2	uimsbf
combined_motion_shape_texture()		
}		
}		

else {		
combined_motion_shape_texture()		
while (nextbits_bytealigned() == resync_marker) {		
video_packet_header()		
combined_motion_shape_texture ()		
}		
}		
next_start_code()		
}		

6.2.5.1 Complexity Estimation Header

read_vop_complexity_estimation_header() {	**No. of bits**	**Mnemonic**
if (estimation_method=='00') {		
if (vop_coding_type==" I") {		
if (opaque) **dcecs_opaque**	8	uimsbf
if (transparent) **dcecs_transparent**	8	uimsbf
if (intra_cae) **dcecs_intra_cae**	8	uimsbf
if (inter_cae) **dcecs_inter_cae**	8	uimsbf
if (no_update) **dcecs_no_update**	8	uimsbf
if (upsampling) **dcecs_upsampling**	8	uimsbf
if (intra_blocks) **dcecs_intra_blocks**	8	uimsbf
if (not_coded_blocks) **dcecs_not_coded_blocks**	8	uimsbf
if (dct_coefs) **dcecs_dct_coefs**	8	uimsbf
if (dct_lines) **dcecs_dct_lines**	8	uimsbf
if (vlc_symbols) **dcecs_vlc_symbols**	8	uimsbf
if (vlc_bits) **dcecs_vlc_bits**	4	uimsbf
if (sadct) **dcecs_sadct**	8	uimsbf
}		
if (vop_coding_type==" P") {		
if (opaque) **dcecs_opaque**	8	uimsbf
if (transparent) **dcecs_transparent**	8	uimsbf
if (intra_cae) **dcecs_intra_cae**	8	uimsbf
if (inter_cae) **dcecs_inter_cae**	8	uimsbf
if (no_update) **dcecs_no_update**	8	uimsbf
if (upsampling) **dcecs_upsampling**	8	uimsbf
if (intra) **dcecs_intra_blocks**	8	uimsbf
if (not_coded) **dcecs_not_coded_blocks**	8	uimsbf
if (dct_coefs) **dcecs_dct_coefs**	8	uimsbf
if (dct_lines) **dcecs_dct_lines**	8	uimsbf
if (vlc_symbols) **dcecs_vlc_symbols**	8	uimsbf
if (vlc_bits) **dcecs_vlc_bits**	4	uimsbf
if (inter_blocks) **dcecs_inter_blocks**	8	uimsbf
if (inter4v_blocks) **dcecs_inter4v_blocks**	8	uimsbf
if (apm) **dcecs_apm**	8	uimsbf
if (npm) **dcecs_npm**	8	uimsbf
if (forw_back_mc_q) **dcecs_forw_back_mc_q**	8	uimsbf

if (halfpel2)	**dcecs_halfpel2**	8	uimsbf
if (halfpel4)	**dcecs_halfpel4**	8	uimsbf
if (sadct)	**dcecs_sadct**	8	uimsbf
if (quarterpel)	**dcecs_quarterpel**	8	uimsbf
}			
if (vop_coding_type==“B”) {			
if (opaque)	**dcecs_opaque**	8	uimsbf
if (transparent)	**dcecs_transparent**	8	uimsbf
if (intra_cae)	**dcecs_intra_cae**	8	uimsbf
if (inter_cae)	**dcecs_inter_cae**	8	uimsbf
if (no_update)	**dcecs_no_update**	8	uimsbf
if (upsampling)	**dcecs_upsampling**	8	uimsbf
if (intra_blocks)	**dcecs_intra_blocks**	8	uimsbf
if (not_coded_blocks)	**dcecs_not_coded_blocks**	8	uimsbf
if (dct_coefs)	**dcecs_dct_coefs**	8	uimsbf
if (dct_lines)	**dcecs_dct_lines**	8	uimsbf
if (vlc_symbols)	**dcecs_vlc_symbols**	8	uimsbf
if (vlc_bits)	**dcecs_vlc_bits**	4	uimsbf
if (inter_blocks)	**dcecs_inter_blocks**	8	uimsbf
if (inter4v_blocks)	**dcecs_inter4v_blocks**	8	uimsbf
if (apm)	**dcecs_apm**	8	uimsbf
if (npm)	**dcecs_npm**	8	uimsbf
if (forw_back_mc_q)	**dcecs_forw_back_mc_q**	8	uimsbf
if (halfpel2)	**dcecs_halfpel2**	8	uimsbf
if (halfpel4)	**dcecs_halfpel4**	8	uimsbf
if (interpolate_mc_q)	**dcecs_interpolate_mc_q**	8	uimsbf
if (sadct)	**dcecs_sadct**	8	uimsbf
if (quarterpel)	**dcecs_quarterpel**	8	uimsbf
}			
if (vop_coding_type==‘S’&& sprite_enable == “static”) {			
if (intra_blocks)	**dcecs_intra_blocks**	8	uimsbf
if (not_coded_blocks)	**dcecs_not_coded_blocks**	8	uimsbf
if (dct_coefs)	**dcecs_dct_coefs**	8	uimsbf
if (dct_lines)	**dcecs_dct_lines**	8	uimsbf
if (vlc_symbols)	**dcecs_vlc_symbols**	8	uimsbf
if (vlc_bits)	**dcecs_vlc_bits**	4	uimsbf
if (inter_blocks)	**dcecs_inter_blocks**	8	uimsbf
if (inter4v_blocks)	**dcecs_inter4v_blocks**	8	uimsbf
if (apm)	**dcecs_apm**	8	uimsbf
if (npm)	**dcecs_npm**	8	uimsbf
if (forw_back_mc_q)	**dcecs_forw_back_q**	8	uimsbf
if (halfpel2)	**dcecs_halfpel2**	8	uimsbf
if (halfpel4)	**dcecs_halfpel4**	8	uimsbf
if (interpolate_mc_q)	**dcecs_interpolate_mc_q**	8	uimsbf
if (quarterpel)	**dcecs_quarterpel**	8	uimsbf
}			
}			

}		

6.2.5.2 Video Plane with Short Header

video_plane_with_short_header() {	No. of bits	Mnemonic
short_video_start_marker	22	bslbf
temporal_reference	8	uimsbf
marker_bit	1	bslbf
zero_bit	1	bslbf
split_screen_indicator	1	bslbf
document_camera_indicator	1	bslbf
full_picture_freeze_release	1	bslbf
source_format	3	bslbf
picture_coding_type	1	bslbf
four_reserved_zero_bits	4	bslbf
vop_quant	5	uimsbf
zero_bit	1	bslbf
do{		
pei	1	bslbf
if (pei == "1")		
psupp	8	bslbf
} while (pei == "1")		
gob_number = 0		
for(i=0; i<num_gobs_in_vop; i++)		
gob_layer()		
if(next_bits() == short_video_end_marker)		
short_video _end_marker	22	uimsbf
while(!bytealigned())		
zero_bit	1	bslbf
}		

gob_layer() {	No. of bits	Mnemonic
gob_header_empty = 1		
if(gob_number != 0) {		
if (next_bits() == gob_resync_marker) {		
gob_header_empty = 0		
gob_resync_marker	17	bslbf
gob_number	5	uimsbf
gob_frame_id	2	bslbf
quant_scale	5	uimsbf
}		
}		

for(i=0; i<num_macroblocks_in_gob; i++)
macroblock()
if(next_bits() != gob_resync_marker &&
nextbits_bytealigned() == gob_resync_marker)		

while(!bytealigned())		

zero_bit	1	bslbf
gob_number++		
}		

video_packet_header() {	**No. of bits**	**Mnemonic**
next_resync_marker()		
resync_marker	17-23	uimsbf
if (video_object_layer_shape != “rectangular”) {		
header_extension_code	1	bslbf
if (header_extension_code && !(sprite_enable = “static” && vop_coding_type == “I”)) {		
vop_width	13	uimsbf
marker_bit	1	bslbf
vop_height	13	uimsbf
marker_bit	1	bslbf
vop_horizontal_mc_spatial_ref	13	simsbf
marker_bit	1	bslbf
vop_vertical_mc_spatial_ref **marker_bit** }	13 1	simsbf bslbf

}
macroblock_number
if (video_object_layer_shape != “binary only”)
	1-14	vlclbf

quant_scale	5	uimsbf
if (video_object_layer_shape == “ rectangular”)		
header_extension_code	1	bslbf
if (header_e xtension_code) {		
do {		
modulo_time_base	1	bslbf
} while (modulo_time_base != ‘0’)		
marker_bit	1	bslbf
vop_time_increment	1-16	bslbf
marker_bit	1	bslbf
vop_coding_type	2	uimsbf
if (video_object_layer_shape != “rectangular”) {		
change_conv_ratio_disable	1	bslbf
if (vop_coding_type != “ I”)		
vop_shape_coding_type	1	bslbf
}		
if (video_object_layer_shape != “binary only”) {		
intra_dc_vlc_thr	3	uimsbf

if (sprite_enable == "GMC" && vop_coding_type == "S" && no_of_sprite_warping_points > 0)		
sprite_trajectory()		
if ((reduced_resolution_vop_enable) && (video_object_layer_shape == "rectangular") && ((vop_coding_type == "P") || (vop_coding_type == "I"))) **vop_reduced_resolution**	1	bslbf
if (vop_coding_type != "I") **vop_fcode_forward**	3	uimsbf
if (vop_coding_type == "B") **vop_fcode_backward**	3	uimsbf
}		
}		
if (newpred_enable) {		
vop_id	4-15	uimsbf
vop_id_for_prediction_indication	1	bslbf
if (vop_id_for_prediction_indication)		
vop_id_for_prediction	4-15	uimsbf
marker_bit	1	bslbf
}		
}		

6.2.5.3 Motion Shape Texture

motion_shape_texture() {	No. of bits	Mnemonic

if (data_partitioned)
data_partitioned_motion _shape_texture()
else
combined_motion_shape_texture()		

}		

combined_motion_shape_texture() {	No. of bits	Mnemonic
do{		
macroblock()		
} while (nextbits_bytealigned() != resync_marker && nextbits_bytealigned() != '000 0000 0000 0000 0000 0000')		
}		

data_ partitioned_motion_shape_texture() {	No. of bits	Mnemonic
if (vop_coding_type == "I") {		
data_partitioned_i_vop()		
} else if (vop_coding_type == "P" || (vop_coding_type == "S" && sprite_enable == "GMC")) {		
data_partitioned_p_vop()		
} else if (vop_coding_type == "B") {		
combined_motion_shape_texture()		

}		
NOTE: Data partitioning is not supported in B-VOPs.		

data_partitioned_i_vop() {	No. of bits	Mnemonic
do{		
if (video_object_layer_shape != “rectangular”){		
bab_type	1-3	vlclbf
if (bab_type >= 4) {		
if (!change_conv_rate_disable)		
conv_ratio	1-2	vlclbf
scan_type	1	bslbf
binary_arithmetic_code()		
}		
}		
if (!transparent_mb()) {		
if (video_object_layer_shape != “rectangle”) {		
do {		
mcbpc	1-9	vlclbf
} while (derived_mb_type == “ stuffing”)		
} else {		
mcbpc	1-9	vlclbf
if (derived_mb_type == “ stuffing”)		
continue		
}		
if (mb_type == 4)		
dquant	2	bslbf
if (use_intra_dc_vlc) {		
for (j = 0; j < 4; j++) {		
if (!transparent_block(j)) {		
dct_dc_size_luminance	2-11	vlclbf
if (dct_dc_size_luminance > 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_luminance > 8)		
marker_bit }	1	bslbf
}		

for (j = 0; j < 2; j++) {
dct_dc_size_chrominance
if (dct_dc_size_chrominance > 0)
	2-12	vlclbf

dct_dc_differential	1-12	vlclbf
if (dct_dc_size_chrominance > 8)		
marker_bit	1	bslbf
}		
}		
}		

} while (next_bits() != dc_marker)		
dc_marker /* 110 1011 0000 0000 0001 */	19	bslbf
for (i = 0; i < mb_in_video_packet; i++) {		
if (!transparent_mb()) {		
ac_pred_flag	1	bslbf
cbpy	1-6	vlclbf
}		
}		
for (i = 0; i < mb_in_video_packet; i++) {		
if (!transparent_mb()) {		
for (j = 0; j < block_count; j++)		
block(j)		
}		
}		
}		

NOTE 1: The value of mb_in_video_packet is the number of macroblocks in a video packet. The count of stuffing macroblocks is not included in this value.

NOTE 2: The value of block_count is 6 in the 4:2:0 format.

NOTE 3: The value of alpha_block_count is 4.

data_partitioned_p_vop() {	**No. of bits**	**Mnemonic**
do{		
if (video_object_layer_shape != “rectangular”){		
bab_type	1-7	vlclbf
if ((bab_type == 1) || (bab_type == 6)) {		
mvds_x	1-18	vlclbf
mvds_y	1-18	vlclbf
}		
if (bab_type >= 4) {		
if (!change_conv_rate_disable)		
conv_ratio	1-2	vlclbf
scan_type	1	bslbf
binary_arithmetic_code()		
}		
}		
if (!transparent_mb()) {		
if (video_object_layer_shape != “rectangle”) {		
do {		
not_coded	1	bslbf
if (!not_coded)		
mcbpc	1-9	vlclbf
} while (!(not_coded || derived_mb_type != “ stuffing”))		
} else {		
not_coded	1	bslbf
if (!not_coded){		
mcbpc	1-9	vlclbf
if (derived_mb_type == “ stuffing”)		
continue		

}		
}		
if (!not_coded) {		
if (sprite_enable == "GMC" && vop_coding_type == "S" && derived_mb_type < 2)		
mcsel	1	bslbf
if ((!(sprite_enable == "GMC" && vop_coding_type == "S" && mcsel) && derived_mb_type < 2) || derived_mb_type == 2)		
motion_coding("forward" , derived_mb_type)		
}		
}		
} while (next_bits() != motion_marker)		
motion_marker /* 1 1111 0000 0000 0001 */	17	bslbf
for (i = 0; i < mb_in_video_packet; i++) {		
if (!transparent_mb()) {		
if (!not_coded){		
if (derived_mb_type >= 3)		
ac_pred_flag	1	bslbf
cbpy	1-6	vlclbf
if (derived_mb_type == 1 || derived_mb_type == 4)		
dquant	2	bslbf
if (derived_mb_type >= 3 && use_intra_dc_vlc) {		
for (j = 0; j < 4; j++) {		
if (!transparent_block(j)) {		
dct_dc_size_luminance	2-11	vlclbf
if (dct_dc_size_luminance > 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_luminance > 8)		
marker_bit	1	bslbf
}		
}		
for (j = 0; j < 2; j++) {		
dct_dc_size_chrominance	2-12	vlclbf
if (dct_dc_size_chrominance > 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_chrominance > 8)		
marker_bit	1	bslbf
}		
}		
}		
}		
for (i = 0; i < mb_in_video_packet; i++) {		
if (!transparent_mb()) {		
if (! not_coded) {		
for (j = 0; j < block_count; j++)		

block(j)		
}		
}		
}		
}		

NOTE 1: The value of mb_in_video_packet is the number of macroblocks in a video packet. The count of stuffing macroblocks is not included in this value.
NOTE 2: The value of block_count is 6 in the 4:2:0 format.
NOTE 3: The value of alpha_block_count is 4.

motion_coding(mode, type_of_mb) {	**No. of bits**	**Mnemonic**
motion_vector(mode)		
if (type_of_mb == 2) {		
for (i = 0; i < 3; i++)		
motion_vector(mode)		
}		
}		

6.2.5.4 Sprite coding

decode_sprite_piece() {	**No. of bits**	**Mnemonic**
piece_quant	5	bslbf
piece_width	9	bslbf
piece_height	9	bslbf
marker_bit	1	bslbf
piece_xoffset	9	bslbf
piece_yoffset	9	bslbf
sprite_shape_texture()		
}		

sprite_shape_texture() {	**No. of bits**	**Mnemonic**
if (sprite_transmit_mode == “piece”) {		
for (i=0; i < piece_height; i++) {		
for (j=0; j < piece_width; j++) {		
if (!send_mb()) {		
macroblock()		
}		
}		
}		
}		
if (sprite_transmit_mode == “update”) {		
for (i=0; i < piece_height; i++) {		
for (j=0; j < piece_width; j++) {		
macroblock()		
}		
}		

}		
}		

sprite_trajectory() {	**No. of bits**	**Mnemonic**
for (i=0; i < no_of_sprite_warping_points; i++) {		
warping_mv_code(du[i])		
warping_mv_code(dv[i])		
}		
}		

warping_mv_code(d) {	**No. of bits**	**Mnemonic**
dmv_length	2-12	uimsbf
if (dmv_length != '00')		
dmv_code	1-14	uimsbf
marker_bit	1	bslbf
}		

brightness_change_factor() {	**No. of bits**	**Mnemonic**
brightness_change_factor_size	1-4	uimsbf
brightness_change_factor_code	5-10	uimsbf
}		

6.2.6 Macroblock

macroblock() {	**No. of bits**	**Mnemonic**
if (vop_coding_type != " B") {		
if (video_object_layer_shape != "rectangular"		
&& !(sprite_enable == "static" && low_latency_sprite_enable		
&& sprite_transmit_mode == " update"))		
mb_binary_shape_coding()		
if (video_object_layer_shape != "binary only") {		
if (!transparent_mb()) {		
if (video_object_layer_shape != "rectangular"		
&& !(sprite_enable = "static" && low_latency_sprite_enable		
&& sprite_transmit_mode == " update")) {		
do{		
if (vop_coding_type != " I" && !(sprite_enable == "static"		
&& sprite_transmit_mode == "piece"))		
not_coded	1	bslbf

if (!not_coded || vop_coding_type == “I” || (vop_coding_type == "S" && low_latency_sprite_enable && sprite_transmit_mode == "piece"))		
mcbpc	1-9	vlclbf
} while(!(not_coded || derived_mb_type != “ stuffing”))		
} else {		
if (vop_coding_type != “I” && !(sprite_enable = "static" && sprite_transmit_mode == “ piece”))		
not_coded	1	bslbf
if (!not_coded || vop_coding_type == “I” || (vop_coding_type == "S" && low_latency_sprite_enable && sprite_transmit_mode == "piece")) **mcbpc**	1-9	vlclbf
}		

if (!not_coded || vop_coding_type == “I”
|| (vop_coding_type == "S" && low_latency_sprite_enable
&& sprite_transmit_mode == "piece")) {		

if (vop_coding_type == “S” && sprite_enable == “GMC” && (derived_mb_type == 0 || derived_mb_type == 1)) **mcsel**	1	bslbf
if (!short_video_header && (derived_mb_type == 3 || derived_mb_type == 4)) **ac_pred_flag**	1	bslbf
if (derived_mb_type != “ stuffing”)		
cbpy	1-6	vlclbf
else		
return()		
if (derived_mb_type == 1 || derived_mb_type == 4) **dquant**	2	bslbf
if (interlaced)		
interlaced_information()		
if (!(ref_select_code==‘ 11’ && scalability) && sprite_enable != “static”) {		

if ((derived_mb_type == 0 || derived_mb_type == 1)
&& (vop_coding_type == “P”
|| (vop_coding_type == “S” && !mcsel))) {
motion_vector(“forward”)
if (interlaced && field_prediction)		

motion_vector(“forward”)		
}		
if (derived_mb_type == 2) {		
for (j=0; j < 4; j++)		
if (!transparent_block(j))		
motion_vector(“forward”)		
}		
}		

for (i = 0; i < block_count; i++)		
if(!transparent_block(i))		
block(i)		
}		
}		
}		
}		
else {		
if (video_object_layer_shape != “rectangular”)		
mb_binary_shape_coding()		
if ((co_located_not_coded != 1 || (scalability && (ref_select_code != '11' || enhancement_type == 1)) || (sprite_enable == “GMC” && backward_reference_vop_coding_type == “S”)) && video_object_layer_shape != “binary only”) {		
if (!transparent_mb()) {		
modb	1-2	vlclbf
if (modb != ‘1’) {		
mb_type	1-4	vlclbf
if (modb == ‘00’)		
cbpb	3-6	vlclbf
if (ref_select_code != ‘00’ || !scalability) {		
if (mb_type != “1” && cbpb!=0)		

dbquant
if (interlaced)
interlaced_information()
if (mb_type == ‘01’ ||
mb_type == ‘ 0001’) {
	1-2	vlclbf

motion_vector(“ forward”)		
if (interlaced && field_prediction)		
motion_vector(“ forward”)		
}		
if (mb_type == ‘01’ || mb_type == ‘ 001’) {		
motion_vector(“ backward”)		
if (interlaced && field_prediction)		
motion_vector(“ backward”)		
}		
if (mb_type == “1”)		
motion_vector(“ direct”)		
}		
if (ref_ select_code == ‘00’ && scalability && cbpb !=0) { **dbquant**	1-2	vlclbf
if (mb_type == ‘01’ || mb_type == ‘1’)		
motion_vector(“ forward”)		
}		
for (i = 0; i < block_count; i++)		
if(!transparent_block(i))		

block(i)		
}		
}		
}		
}		
if(video_object_layer_shape==“grayscale”		
&& !transparent_mb()) {		
for(j=0; j<aux_comp_count; j++) {		
if(vop_coding_type==”I” || (vop_coding_type==” P” ||		
(vop_coding_type==”S” && sprite_enable==”GMC”))		
&& !not_coded &&		
(derived_mb_type==3 || derived_mb_type==4))) {		
coda_i	1	bslbf
if(coda_i==”coded”) {		
ac_pred_flag_alpha	1	bslbf
cbpa	1-6	vlclbf
for(i=0;i<alpha_block_count;i++)		
if(!transparent_block())		
alpha_block(i)		
}		
} else { /* P, S(GMC) or B macroblock */		
if(vop_coding_type == ”P” || (sprite_enable == “ GMC” &&		
(vop_coding_type==“ S”		
|| backward_reference_vop_coding_type==“S”))		
|| co_located_not_coded != 1) {		
coda_pb	1-2	vlclbf
if(coda_pb==” coded”) {		
cbpa	1-6	vlclbf
for(i=0;i<alpha_block_count;i++)		
if(!transparent_block())		
alpha_block(i)		
}		
}		
}		
}		
}		
NOTE: The value of block_count is 6 in the 4:2:0 format. The value of alpha_block_count is 4. backward_reference_vop_coding_type means the vop_coding_type of the backward reference VOP as described in subclause 7.6.7.		

6.2.6.1 MB Binary Shape Coding

mb_binary_shape_coding() {	**No. of bits**	**Mnemonic**
if(!(scalability && hierarchy_type == ‘0’		
&& (enhancement_type == ‘0’ || use_ref_shape == ‘0’))		
&& !(scalability && video_object_layer_shape == “ binary only”)) {		
bab_type	1-7	vlclbf

if (vop_coding_type == 'P' || vop_coding_type == 'B' || (vop_coding_type == 'S' && sprite_enable == "GMC")) {		
if ((bab_type==1) || (bab_type == 6)) {		
mvds_x	1-18	vlclbf
mvds_y	1-18	vlclbf
}		
}		
if (bab_type >=4) {		

if (!change_conv_ratio_disable)
conv_ratio
scan_type
binary_arithmetic_code()
}
	
	1-2
	1
		
		vlcbf
		bslbf

} else {		
if (!use_ref_shape || video_object_layer_shape == "binary only") {		
enh_bab_type	1-3	vlclbf
if (enh_bab_type == 3)		
scan_type	1	bslbf
if (enh_bab_type == 1 || enh_bab_type == 3)		
enh_binary_arithmetic_code()		
}		
}		
}		

backward_shape () {	**No. of bits**	**Mnemonic**
for(i=0; i<backward_shape_height/16; i++)		
for(j=0; j<backward_shape_width/16; j++) {		
bab_type	1-3	vlclbf
if (bab_type >=4) {		
if (!change_conv_ratio_disable)		
conv_ratio	1-2	vlcbf
scan_type	1	bslbf
binary_arithmetic_code()		
}		
}		
}		

forward_shape () {	**No. of bits**	**Mnemonic**
for(i=0; i<forward_shape_height/16; i++)		
for(j=0; j<forward_shape_width/16; j++) {		
bab_type	1-3	vlclbf
if (bab_type >=4) {		
if (!change_conv_ratio_disable)		
conv_ratio	1-2	vlcbf
scan_type	1	bslbf

binary_arithmetic_code() }		
}		
}		

6.2.6.2 Motion vector

motion_vector (mode) {	No. of bits	Mnemonic
if (mode == „direct“) {		
horizontal_mv_data	1-13	vlclbf
vertical_mv_data	1-13	vlclbf
}		
else if (mode == „forward“) {		
horizontal_mv_data	1-13	vlclbf

if ((vop_fcode_forward != 1) && (horizontal_mv_data != 0))
horizontal_mv_residual
vertical_mv_data
if ((vop_fcode_forward != 1) && (vertical_mv_data != 0))
vertical_mv_residual
	
	1-6
	1-13
	
	1-6
		
		uimsbf
		vlclbf
		
		uimsbf

}		
else if (mode == „backward“) {		
horizontal_mv_data	1-13	vlclbf
if ((vop_fcode_backward != 1) && (horizontal_mv_data != 0))		
horizontal_mv_residual	1-6	uimsbf
vertical_mv_data	1-13	vlclbf
if ((vop_fcode_backward != 1) && (vertical_mv_data != 0))		
vertical_mv_residual	1-6	uimsbf
}		
}		

6.2.6.3 Interlaced Information

interlaced_information( ) {	No. of bits	Mnemonic
if ((derived_mb_type == 3) || (derived_mb_type == 4) || (cbp != 0))		
dct_type	1	bslbf

if (((vop_coding_type == “P”) &&
((derived_mb_type == 0) || (derived_mb_type == 1))) ||
((sprite_enable == “ GMC”) && (vop_coding_type == “S”) &&
(derived_mb_type < 2) && (!mcsel)) ||
((vop_coding_type == “B”) && (mb_type != “1”))) {		

field_prediction	1	bslbf
if (field_prediction) {		

if (vop_coding_type == “P” ||
(vop_coding_type == “B” &&
mb_type != “001”)) {		

forward_top_field_reference	1	bslbf

forward_bottom_field_reference	1	bslbf
}		
if ((vop_coding_type == "B") && (mb_type != "0001")) {		
backward_top_field_reference	1	bslbf
backward_bottom_field_reference	1	bslbf
}		
}		
}		
}		

6.2.7 Block

The detailed syntax for the term "DCT coefficient" is fully described in clause 7.

block(i) {	**No. of bits**	**Mnemonic**
last = 0		
if(!data_partitioned && (derived_mb_type == 3 || derived_mb_type == 4)) {		
if(short_video_header == 1)		
intra_dc_coefficient	8	uimsbf
else if (use_intra_dc_vlc == 1) {		
if (i<4) {		
dct_dc_size_luminance	2-11	vlclbf
if(dct_dc_size_luminance != 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_luminance > 8)		
marker_bit	1	bslbf
} else {		
dct_dc_size_chrominance	2-12	vlclbf
if(dct_dc_size_chrominance !=0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_chrominance > 8)		
marker_bit	1	bslbf
}		
}		
}		
if (pattern_code[i])		
while (! last)		
DCT coefficient	3-24	vlclbf
}		
NOTE : "last" is defined to be the LAST flag resulting from reading the most recent DCT coefficient.		

6.2.7.1 Alpha Block

The syntax for DCT coefficient decoding is the same as for block(i) in subclause 6.2.8.

alpha_block(i) {	No. of bits	Mnemonic
last = 0		
if(!data_partitioned && (vop_coding_type == "I" || (vop_coding_type == "P" || (vop_coding_type == "S" && sprite_enable == "GMC")) && !not_coded && (derived_mb_type == 3 || derived_mb_type == 4))) {		
dct_dc_size_alpha	2-11	vlclbf
if(dct_dc_size_alpha != 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_alpha > 8)		
marker_bit	1	bslbf
}		
if (pattern_code[i])		
while (! last)		
DCT coefficient	3-24	vlclbf
}		
NOTE: "last" is defined to be the LAST flag resulting from reading the most recent DCT coefficient.		

6.2.8 Still Texture Object

StillTextureObject() {	No. of bits	Mnemonic
still_texture_object_start_code	32	

if (visual_object_verid != 0001) {		
tiling_disable	1	bslbf
texture_error_resilience_disable	1	bslbf
texture_object_id	16	uimsbf
marker_bit	1	bslbf
wavelet_filter_type	1	uimsbf
wavelet_download	1	uimsbf
wavelet_decomposition_levels	4	uimsbf
scan_direction	1	bslbf
start_code_enable	1	bslbf
texture_object_layer_shape	2	uimsbf
quantisation_type	2	uimsbf
if (quantisation_type == 2) {		
spatial_scalability_levels	4	uimsbf
if (spatial_scalability_levels != wavelet_decomposition_levels) {		
use_default_spatial_scalability	1	uimsbf
if (use_default_spatial_layer_size == 0)		
for (i=0; i<spatial_scalability_levels – 1; i++)		
wavelet_layer_index	4	uimsbf
}		
}		
if (wavelet_download == “1”){		
uniform_wavelet_filter	1	uimsbf
if (uniform_wavelet_filter == “1”)		
download_wavelet_filters()		
else		
for (i=0; i<wavelet_decomposition_levels; i++)		
download_wavelet_filters( )		
}		
wavelet_stuffing	3	uimsbf
if(!texture_error_resilience_disable) {		
target_segment_length	16	uimsbf

marker_bit	1	bslbf
}		
if(texture_object_layer_shape == "00") {		
texture_object_layer_width	15	uimsbf
marker_bit	1	bslbf
texture_object_layer_height	15	uimsbf
marker_bit	1	bslbf
}		
else if (texture_object_layer_shape == "01") {		
horizontal_ref	15	uimsbf
marker_bit	1	bslbf
vertical_ref	15	uimsbf
marker_bit	1	bslbf
object_width	15	uimsbf
marker_bit	1	bslbf
object_height	15	uimsbf
marker_bit	1	bslbf
/* if tiling_disable == "1" and texture_object_layer_shape == "01" configuration information precedes this point; elementary stream data follows. See annex K */		
if(tiling_disable == "1")		
shape_object_decoding()		
}		
if (tiling_disable == "0"){		
tile_width	15	uimsbf
marker_bit	1	bslbf
tile_height	15	uimsbf
marker_bit	1	bslbf
number_of_tiles	16	uimsbf
marker_bit	1	bslbf
tiling_jump_table_enable	1	bslbf
if (tiling_jump_table_enable == "1"){		
for (i=0; i<number_of_tiles; i++) {		
tile_size_high	16	uimsbf
market_bit	1	bslbf
tile_size_low	16	uimsbf
market_bit	1	bslbf
}		
}		
next_start_code()		
}		
/* if tiling_disable == "0" or texture_object_layer_shape == "00" configuration information precedes this point; elementary stream data follows. See annex K */		
do {		
if(tiling_disable == "0") {		
texture_tile_start_code	32	bslbf
tile_id	16	uimsbf
if (texture_object_layer_shape == "01") {		
marker_bit	1	bslbf
texture_tile_type	2	uimsbf

marker_bit	1	bslbf
}		
}		
if (!texture_error_resilience_disable) {		
if (texture_object_layer_shape== “ 01” && tiling_disable==“ 0”) {		
if (texture_tile_type==“boundary tile”)		
shape_object_decoding()		
}		
while (nextbit_bytealigned () == texture_marker) {		
TexturePacketHeader ()		
do {		
while (texture_unit_not_completed) {		
DecodeStu()		
if (segment_length>= target_segment_length)		
decode_segment_marker()		
}		
} while (nextbit_bytealigned () != texture_ma rker)		
}		
}		
else {		
if (texture_object_layer_shape== “ 01” && tiling_disable==“0”) {		
if (texture_tile_type==“boundary tile”) shape_object_decoding() } for (color = “y”, “u”, “v”) { wavelet_dc_decode() }		
if (quantisation_type == 1){		
TextureLayerSQ ( )		
}		
else if (quantisation_type== 2) {		
if (start_code_enable == 1) {		
do {		
TextureSpatialLayerMQ()		
} while (next_bits() == texture_spatial_layer_start_code)		
} else {		
for (i =0; i<spatial_scalability_levels; i++)		
TextureSpatialLayerMQNSC()		
}		
} else if (quantisation_type == 3) { for (color = “y”, “u”, “v”) do { **quant_byte** } while (quant_byte >> 7)	8	uimsbf

max_bitplanes	5	uimsbf
if (scan_direction == 0) {		
do {		
TextureSNRLayerBQ ( )		
} while (next_bits() == texture_snr_layer_start_code)		
} else {		
do {		
TextureSpatialLayerBQ ( )		
} while (next_bits() == texture_spatial_layer_start_code)		
}		
}		
} /* error_resi_disable */		
If (tiling_disable == “0”)		
next_start_code()		
} while (nextbits_bytealigned () == texture_tile_start_code)		
}		
else { /* version 1 */		

texture_object_id	16	uimsbf
marker_bit	1	bslbf
wavelet_filter_type	1	uimsbf
wavelet_download	1	uimsbf
wavelet_decomposition_levels	4	uimsbf
scan_direction	1	bslbf
start_code_enable	1	bslbf
texture_object_layer_shape	2	uimsbf
quantization_type	2	uimsbf
if (quantization_type == 2) {		
spatial_scalability_levels	4	uimsbf
if (spatial_scalability_levels != wavelet_decomposition_levels) {		
use_default_spatial_scalability	1	uimsbf

if (use_default_spatial_layer_size == 0)
for (i=0; i<spatial_scalability_levels – 1; i++)
wavelet_layer_index
}
}
if (wavelet_download == "1"){
	4	

uniform_wavelet_filter	1	uimsbf
if (uniform_wavelet_filter == "1")		
download_wavelet_filters()		
else		
for (i=0; i<wavelet_decomposition_levels; i++)		
download_wavelet_filters()		
}		
wavelet_stuffing	3	uimsbf
if(texture_object_layer_shape == " 00"){		
texture_object_layer_width	15	uimsbf
marker_bit	1	bslbf
texture_object_layer_height	15	uimsbf
marker_bit	1	bslbf
}		
else {		
horizontal_ref	15	imsbf
marker_bit	1	bslbf
vertical_ref	15	imsbf
marker_bit **object_width** **marker_bit** **object_height** **marker_bit** shape_object_decoding ( )	1 15 1 15 1	bslbf uimsbf bslbf uimsbf bslbf
}		
/* configuration information precedes this point; elementary stream data follows. See annex K */		
for (color = "y", "u", "v")		

wavelet_dc_decode()	
if(quantization_type == 1)	
TextureLayerSQ()	
else if (quantization_type == 2) {	
if (start_code_enable == 1) {	
do {	
TextureSpatialLayerMQ()	
} while (next_bits() == texture_spatial_layer_start_code)	
} else {	
for (i =0; i<spatial_scalability_levels; i++)	
TextureSpatialLayerMQNSC()	
}	
}	
else if (quantization_type == 3) {	
for (color = “y”, “u”, “v”)	
do{	
quant_byte	
} while(quant_byte >>7)	
max_bitplanes	
if (scan_direction == 0) {	
do { TextureSNRLayerBQ() } while (next_bits() == texture_snr_layer_start_code)	

} else {
do {
TextureSpatialLayerBQ()	

} while (next_bits() == texture_spatial_layer_start_code)	
}	
}	
}	
}	
NOTE 1 : The value of texture_unit_not_completed is '0' if the decoding of one sub-unit in a texture unit is completed. Otherwise the value is '1'.	
NOTE 2 : The value of first_packet_decoded is set to ‘0’ if the first packet has not been decoded. The value of first_packet_decoded is set to ‘1’ after the first packet has been decoded.	

TexturePacketHeader() {	No. of bits	Mnemonic
next_texture_marker()		
texture_marker	17	bslbf
do {		
TU_first	8	uimsbf
} while (TU_first >> 7)		
do{		
TU_last	8	uimsbf
} while (TU_last >> 7)		
header_extention_code	1	bslbf
if (header_extention_code) {		
texture_object_id	16	uimsbf
marker_bit	1	bslbf
wavelet_filter_type	1	uimsbf
wavelet_download	1	uimsbf
wavelet_decomposition_levels	4	uimsbf
scan_direction	1	bslbf
start_code_enable	1	bslbf
texture_object_layer_shape	2	uimsbf
quantisation_type	2	uimsbf
if (quantisation_type == 2) {		
spatial_scalability_levels	4	uimsbf
if (spatial_scalability_levels != wavelet_decomposition_levels) {		
use_default_spatial_scalability	1	uimsbf
if (use_default_spatial_layer_size == 0) for (i=0; i<spatial_scalability_levels – 1; i++) **wavelet_layer_index**	4	uimsbf
}		
}		
if (wavelet_download == "1"){		
uniform_wavelet_filter	1	uimsbf
if (uniform_wavelet_filter == "1")		
download_wavelet_filters()		
else		
for (I=0; i<wavelet_decomposition_levels; i++)		
download_wavelet_filters( )		
}		
wavelet_stuffing	3	uimsbf
if(texture_object_layer_shape == "00") {		
texture_object_layer_width	15	uimsbf
marker_bit	1	bslbf
texture_object_layer_height	15	uimsbf
marker_bit	1	bslbf
}		
else {		
if (!first_packet_decoded) {		
horizontal_ref	15	uimsbf

marker_bit	1	bslbf
vertical_ref	15	uimsbf
marker_bit	1	bslbf
object_width	15	uimsbf
marker_bit	1	bslbf
object_height	15	uimsbf
marker_bit	1	bslbf
}		
}		
if (tiling_disable == "0"){		
tile_width	15	uimsbf
marker_bit	1	bslbf
tile_height	15	uimsbf
marker_bit	1	bslbf
}		
target_segment_length	16	uimsbf
}		
}		

DecodeStu() {	No. of bits	Mnemonic
for (color = " y", "u", "v")		
wavelet_dc_decode ()		
if(quantisation_type == 1) {		
TextureLayerSQ()		
}		
else if (quantisation_type == 2) {		
if (start_code_enable == 1) {		
do {		
TextureSpatialLayerMQ()		
} while (next_bits() == texture_spati al_layer_start_code)		
} else {		
for (i =0; i<spatial_scalability_levels; i++)		
TextureSpatialLayerMQNSC()		
}		
}		
else if (quantisation_type == 3) {		
for (color = " y", "u", "v")		
do{		
quant_byte	8	uimsbf
} while(quant_byte >>7)		
max_bitplanes	5	uimsbf
if (scan_direction == 0) {		
do {		
TextureSNRLayerBQ()		
} while (next_bits() == texture_snr_layer_start_code)		
} else {		
do {		
TextureSpatialLayerBQ()		
} while (next_bits() == texture_spatial_layer_start_code)		
}		
}		
}		

6.2.8.1 TextureLayerSQ

TextureLayerSQ() {	No. of bits	Mnemonic
if (scan_direction == 0) {		
for ("y", "u", " v") {		
do {		
quant_byte	8	uimsbf
} while (quant_byte >> 7)		

for (i=0; i<wavelet_decomposition_levels; i++)
if (i!=0 || color!= "u"," v"){
max_bitplane[i]
if ((i+1)%4==0)
	5	uimsbf

marker_bit	1	bslbf
}		
}		
for (i = 0; i<tree_blocks; i++)		
for (color = "y","u","v")		
arith_decode_highbands_td()		
} else {		
if (start_code_enable) {		
do {		
TextureSpatialLayerSQ()		
} while (next_bits() == texture_spatial_layer_start_code)		
} else {		
for (i = 0; i< wavelet_decomposition_levels; i++)		
TextureSpatialLayerSQNSC()		
}		
}		
}		
NOTE: The value of tree_block is that wavelet coefficients are organized in a tree structure which is rooted in the low-low band (DC band) of the wavelet decomposition, then extends into the higher frequency bands at the same spatial location. Note the DC band is encoded separately.		

6.2.8.2 TextureSpatialLayerSQ

TextureSpatialLayerSQ() {	**No. of bits**	**Mnemonic**
texture_spatial_layer_start_code	32	bslbf
texture_spatial_layer_id	5	uimsbf
TextureSpatialLayerSQNSC()		
}		

6.2.8.3 TextureSpatialLayerSQNSC

TextureSpatialLayerSQNSC() {	**No. of bits**	**Mnemonic**
for (color="y","u","v"){		
if ((first_wavelet_layer && color==" y") || (second_wavelet_layer && color=="u","v"))		
do {		
quant_byte	8	uimsbf
} while (quant_byte >> 7)		
if (color =="y")		
max_bitplanes	5	uimbsf
else if (!first_wavelet_layer)		
max_bitplanes	5	uimbsf
}		
for (color="y","u","v")		
if (color="y" || !first_wavelet_layer)		
arith_decode_highbands_bb()		
}		

NOTE:The value of first_wavelet_layer becomes "true" when the variable 'i' of subclause 6.2.8.1 TextureLayerSQ() equals to zero. Otherwise, it is " false" . The value of second_wavelet_layer becomes " true" when the variable 'i' of sublause 6.2.8.1 TextureLayerSQ() equals to one. Otherwise, it is " false" .

6.2.8.4 TextureSpatialLayerMQ

TextureSpatialLayerMQ() {	No. of bits	Mnemonic
texture_spatial_layer_start_code	32	bslbf
texture_spatial_layer_id	5	uimsbf
snr_scalability_levels	5	uimsbf
do {		
TextureSNRLayerMQ( )		
} while (next_bits() == texture_snr_layer_start_code)		
}		

6.2.8.5 TextureSpatialLayerMQNSC

TextureSpatialLayerMQNSC() {	No. of bits	Mnemonic
snr_scalability_levels	5	uimsbf
for (i =0; i<snr_scalability_levels; i++)		
TextureSNRLayerMQNSC ( )		
}		

6.2.8.6 TextureSNRLayerMQ

TextureSNRLayerMQ(){		
texture_snr_layer_start_code	32	bslbf
texture_snr_layer_id	5	uimsbf
TextureSNRLayerMQNSC()		
}		

6.2.8.7 TextureSNRLayerMQNSC

TextureSNRLayerMQNSC(){	No. of bits	Mnemonic
if (spatial_scalability_levels == wavelet_decomposition_levels && spatial_layer_id == 0) {		
for (color = " y") {		
do {		
quant_byte	8	uimsbf
} while (quant_byte >> 7)		
for (i=0; i<spatial_layers; i++) {		
max_bitplane[i]	5	uimsbf
if ((i+1)%4 == 0)		
marker_bit	1	bslbf

}		
}		
}		
else {		
for (color="y", "u", "v") {		
do {		
quant_byte	8	uimsbf
} while (quant_byte >> 7)		
for (i=0; i<spatial_layers; i++) {		
max_bitplane[i]	5	uimsbf
if ((i+1)%4 == 0)		
marker_bit	1	bslbf
}		
}		
}		
if (scan_direction == 0) {		
for (i = 0; i<tree_blocks; i++)		
for (color = "y","u","v")		
if (wavelet_decomposition_layer_id != 0 || color != "u", "v")		
arith_decode_highbands_td()		
} else {		
for (i = 0; i< spatial_layers; i++) {		
for (color = "y","u","v"){		
if (wavelet_decomposition_layer_id != 0 || color != "u", "v")		
arith_decode_highbands_bb()		
}		
}		
}		
}		
NOTE: The value of spatial_layers is equivalent to the maximum number of the wavelet decomposition layers in that scalability layer.		

6.2.8.8 TextureSpatialLayerBQ

TextureSpatialLayerBQ() {	**No. of bits**	**Mnemonic**
texture_spatial_layer_start_code	32	bslbf
texture_spatial_layer_id	5	uimsbf
for (i=0; i<max_bitplanes; i++) {		
texture_snr_layer_start_code	32	bslbf
texture_snr_layer_id	5	uimsbf
TextureBitPlaneBQ()		
next_start_code()		
}		
}		

6.2.8.9 TextureBitPlaneBQ

TextureBitPlaneBQ () {	No. of bits	Mnemonic
for (color = " y", "u", "v")		
if (wavelet_decomposition_layer_id == 0){		
all_nonzero[color]	1	bslbf
if (all_nonzero[color] == 0) {		
all_zero[color]	1	bslbf
if (all_zero[color]==0) {		
lh_zero[color]	1	bslbf
hl_zero[color]	1	bslbf
hh_zero[color]	1	bslbf
}		
}		
}		
if (wavelet_decomposition_layer_id != 0 ||color != "u", "v"){		
if(all_nonzero[color]==1 || all_zero[color]==0){		
if (scan_direction == 0)		
arith_decode_highbands_bilevel_bb()		
else		
arith_decode_highbands_bilevel_td()		
}		
}		
}		
}		

6.2.8.10 TextureSNRLayerBQ

TextureSNRLayerBQ() {	No. of bits	Mnemonic
texture_snr_layer_start_code	32	bslbf
texture_snr_layer_id	5	uimsbf
for (i=0; i<wavelet_decomposition_levels; i++) {		
texture_spatial_layer_start_code	32	bslbf
texture_spatial_layer_id	5	uimsbf
TextureBitPlaneBQ()		
next_start_code ( )		
}		
}		

6.2.8.11 DownloadWaveletFilters

download_wavelet_filters() {	No. of bits	Mnemonic
lowpass_filter_length	4	uimsbf
highpass_filter_length	4	uimsbf
do{		

if (wavelet_filter_type == 0) {		
filter_tap_integer	16	imsbf
marker_bit	1	bslbf
} else {		
filter_tap_float_high	16	uimsbf
marker_bit	1	bslbf
filter_tap_float_low	16	uimsbf
marker_bit	1	bslbf
}		
} while (lowpass_filter_length--)		
do{		
if (wavelet_filter_type == 0){		
filter_tap_integer	16	imsbf
marker_bit	1	bslbf
} else {		
filter_tap_float_high	16	uimsbf
marker_bit	1	bslbf
filter_tap_float_low	16	uimsbf
marker_bit	1	bslbf
}		
} while (highpass_filter_length--)		
if (wavelet_filter_type == 0) {		
integer_scale	16	uimsbf
marker_bit	1	bslbf
}		
}		

6.2.8.12 Wavelet dc decode

wavelet_dc_decode() {	**No. of bits**	**Mnemonic**
mean	8	uimsbf
do{		
quant_dc_byte	8	uimsbf
} while(quant_dc_byte >>7)		
do{		
band_offset_byte	8	uimsbf
} while (band_offset_byte >>7)		
do{		
band_max_byte	8	uimsbf
} while (band_max_byte >>7)		
arith_decode_dc()		
}		

6.2.8.13 Wavelet higher bands decode

wavelet_ higher_bands_decode() {	**No. of bits**	**Mnemonic**
do{		

root_max_alphabet_byte	8	uimsbf
} while (root_max_alphabet_byte >>7)		
marker_bit	1	bslbf
do{		
valz_max_alphabet_byte	8	uimsbf
} while (valz_max_alphabet_byte >>7)		
do{		
valnz_max_alphabet_byte	8	uimsbf
} while (valnz_max_alphabet_byte >>7)		
arith_decode_highbands()		
}		

6.2.8.14 Shape Object Decoding

shape_object_decoding() {	**No. of bits**	**Mnemonic**
change_conv_ratio_disable	1	bslbf
sto_constant_alpha	1	bslbf
if (sto_constant_alpha)		
sto_constant_alpha_value	8	bslbf
if (visual_object_verid != 0001) {		

marker_bit	1	bslbf
for(j=0; j<shape_base_layer_height_blocks(); j++) {		
for(j=0; j<shape_base_layer_width_blocks(); j++) {		
bab_type	1-2	vlclbf
if (bab_type == 4) {		
if (!change_conv_ratio_disable)		
conv_ratio	1-2	vlclbf
scan_type	1	bslbf
binary_arithmetic_decode()		
}		
}		
}		
marker_bit	1	bslbf
if (!start_code_enable) {		
sto_shape_coded_layers	4	uimsbf
marker_bit	1	bslbf
for(k = 0; k < sto_shape_coded_layers; k++) {		
for(i=0;i<shape_enhanced_layer_height_blocks(); i++)		
for(j=0; j<shape_enhanced_layer_width_blocks(); j++)		
enh_binary_arithmetic_decode()		
marker_bit	1	bslbf
}		
}		
else {		
next_start_code()		
while (nextbits() == texture_shape_layer_start_code){		
texture_shape_layer_start_code	32	bslbf
texture_shape_layer_id	5	uimsbf
marker_bit	1	bslbf
for(i=0;i<shape_enhanced_layer_height_blocks(); i++)		
for(j=0; j<shape_enhanced_layer_width_blocks(); j++)		
enh_binary_arithmetic_decode()		
marker_bit	1	bslbf
next_start_code()		
}		
texture_spatial_layer_start_code	32	bslbf
texture_spatial_layer_id	5	uimsbf
marker_bit	1	bslbf
}		
} else { /* version 1 */		

for (i=0 ; i<((object_width+15)/16)*(object_height+15)/16) ; i ++) {		
bab_type	1-2	vlclbf
if (bab_type==4) {		
if (!change_conv_ratio_disable)		
conv_ratio	1-2	vlclbf
scan_type	1	bslbf
binary_arithmetic_decode()		
}		
}		
}		

6.2.9 Mesh Object

MeshObject() {	**No. of bits**	**Mnemonic**
mesh_object_start_code	32	bslbf
do{		
MeshObjectPlane()		
} while (next_bits_bytealigned() == mesh_object_plane_start_code || next_bits_bytealigned() != '0000 0000 0000 0000 0000 0001')		
}		

6.2.9.1 Mesh Object Plane

MeshObjectPlane() {	**No. of bits**	**Mnemonic**
MeshObjectPlaneHeader()		
MeshObjectPlaneData()		
}		

MeshObjectPlaneHeader() {	**No. of bits**	**Mnemonic**
if (next_bits_bytealigned()=='0000 0000 0000 0000 0000 0001'){		
next_start_code()		
mesh_object_plane_start_code	32	bslbf
}		
is_intra	1	bslbf
mesh_mask	1	bslbf
temporal_header()		
}		

MeshObjectPlaneData() {	**No. of bits**	**Mnemonic**
if (mesh_mask == 1) {		
if (is_intra == 1)		
mesh_geometry()		

else		
mesh_motion()		
}		
}		

6.2.9.2 Mesh geometry

mesh_geometry() {	No. of bits	Mnemonic
mesh_type _code	2	bslbf
if (mesh_type_code == '01') {		
nr_of_mesh_nodes_hor	10	uimsbf
nr_of_mesh_nodes_vert	10	uimsbf
marker_bit	1	uimsbf
mesh_rect_size_hor	8	uimsbf
mesh_rect_size_vert	8	uimsbf
triangle_split_code	2	bslbf
}		
else if (mesh_type_code == '10') {		
nr_of_mesh_nodes	16	uimsbf
marker_bit	1	uimsbf
nr_of_boundary_nodes	10	uimsbf
marker_bit	1	uimsbf
node0_x	13	simsbf
marker_bit	1	uimsbf
node0_y	13	simsbf
marker_bit	1	uimsbf
for (n=1; n < nr_of_mesh_nodes; n++) {		
delta_x_len_vlc	2-12	vlclbf
if (delta_x_len_vlc)		
delta_x	1-14	vlclbf
delta_y_len_vlc	2-12	vlclbf
if (delta_y_len_vlc)		
delta_y	1-14	vlclbf
}		
}		
}		

6.2.9.3 Mesh motion

mesh_motion() {	No. of bits	Mnemonic
motion_range_code	3	bslbf
for (n=0; n <nr_of_mesh_nodes; n++) {		
node_motion_vector_flag	1	bslbf
if (node_motion_vector_flag == '0') {		
delta_mv_x_vlc	1-13	vlclbf
if ((motion_range_code != 1) && (delta_mv_x_vlc != 0))		
delta_mv_x_res	1-6	uimsbf

delta_mv_y_vlc	1-13	vlclbf
if ((motion_range_code != 1) && (delta_mv_y_vlc != 0))		
delta_mv_y_res	1-6	uimsbf
}		
}		
}		

6.2.10 FBA Object

fba_object() {	**No. of bits**	**Mnemonic**
fba_object_start_code do { fba_object_plane()	32	bslbf
} while (!((nextbits_bytealigned() == '000 0000 0000 0000 0000 0000') && (nextbits_bytealigned() != fba_object_plane_start_code)))		
}		

6.2.10.1 FBA Object Plane

fba_object_plane() {	**No. of bits**	**Mnemonic**
fba_object_plane_header()		
fba_object_plane_data()		
}		

fba_object_plane_header() {	**No. of bits**	**Mnemonic**
is_intra	1	bslbf
fba_object_mask	2	bslbf
temporal_header()		
}		

fba_object_plane_data() {	**No. of bits**	**Mnemonic**
if(fba_object_mask &'01') {		
if(is_intra) {		
fap_quant	5	uimsbf
for (group_number = 1; group_number <= 10; group_number++) {		
marker_bit	1	uimsbf
fap_mask_type	2	bslbf
if(fap_mask_type == '01'|| fap_mask_type == '10')		
fap_group_mask[group_number]	2-16	vlcbf
}		
fba_suggested_gender	1	bslbf
fba_object_coding_type	1	bslbf

if(fba_object_coding_type == 0) {		
is_i_new_max	1	bslbf
is_i_new_min	1	bslbf
is_p_new_max	1	bslbf
is_p_new_min	1	bslbf
decode_new_minmax()		
decode_ifap()		
}		
if(fba_object_coding_type == 1)		
decode_i_segment()		
}		
else {		
if(fba_object_coding_type == 0)		
decode_pfap()		
if(fba_object_coding_type == 1)		
decode_p_segment()		
}		
}		
}		

if(fba_object_mask &'10') {		
if(is_intra) {		
bap_pred_quant_index	5	uimsbf
for (group_number = 1; group_number <= BAP_NUM_GROUPS; group_number++) {		
marker_bit	1	uimsbf
bap_mask_type	2	bslbf
if(bap_mask_type == '01')		
bap_group_mask [group_number]	3-22	vlcbf
else if (bap_mask_type == '00') {		
for(i=0; i <BAPS_IN_GROUP[group_number];i++) {		
bap_group_mask[group_mask][i] = 0		
}		
}		
else if (bap_mask_type == '11'){		
for(i=0; i <BAPS_IN_GROUP[group_number];i++) {		
bap_group_mask[group_mask][i] = 1		
}		
}		
}		
fba_suggested_gender	1	bslbf
fba_object_coding_type	1	bslbf
if (fba_object_coding_type == 0) {		
bap_is_i_new_max	1	bslbf
bap_is_i_new_min	1	bslbf
bap_is_p_new_max	1	bslbf
bap_is_p_new_min	1	bslbf
decode_bap_new_minmax()		
decode_bap_ibap()		
}		
if(fba_object_coding_type == 1)		
decode_bap_i_segment()		
}		
else {		
if (fba_object_coding_type == 0)		
decode_bap_pbap()		
if(fba_object_coding_type == 1)		
decode_bap_p_segment()		
}		
}		

temporal_header() {	**No. of bits**	**Mnemonic**
if (is_intra) {		
is_frame_rate	1	bslbf
if(is_frame_rate)		
decode_frame_rate()		

is_time_code	1	bslbf
if (is_time_code)		
time_code	18	bslbf
}		
skip_frames	1	bslbf
if(skip_frames)		
decode_skip_frames()		
}		

6.2.10.2 Decode frame rate and skip frames

decode_frame_rate(){	**No. of bits**	**Mnemonic**
frame_rate	8	uimsbf
seconds	4	uimsbf
frequency_offset	1	uimsbf
}		

decode_skip_frames(){	**No. of bits**	**Mnemonic**
do{		
number_of_frames_to_skip	4	uimsbf
} while (number_of_frames_to_skip = " 1111")		
}		

6.2.10.3 Decode new minmax

decode_new_minmax() {	**No. of bits**	**Mnemonic**
if (is_i_new_max) {		
for (group_number = 2, j=0, group_number <= 10, group_number++)		
for (i=0; i < NFAP[group_number]; i++, j++) {		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (fap_group_mask[group_number] & (1 <<i))		
i_new_max[j]	5	uimsbf
}		
if (is_i_new_min) {		
for (group_number = 2, j=0, group_number <= 10, group_number++)		
for (i=0; i < NFAP[group_number]; i++, j++) {		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (fap_group_mask[group_number] & (1 <<i))		
i_new_min[j]	5	uimsbf
}		
if (is_p_new_max) {		
for (group_number = 2, j=0, group_number <= 10, group_number++)		
for (i=0; i < NFAP[group_number]; i++, j++) {		
if (!(i & 0x3))		

marker_bit	1	uimsbf
if (fap_group_mask[group_number] & (1 <<i))		
p_new_max[j]	5	uimsbf
}		
if (is_p_new_min) {		
for (group_number = 2, j=0, group_number <= 10, group_number++)		
for (i=0; i < NFAP[group_number]; i++, j++) {		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (fap_group_mask[group_number] & (1 <<i))		
p_new_min[j]	5	uimsbf
}		
}		
}		

6.2.10.4 Decode ifap

decode_ifap(){	**No. of bits**	**Mnemonic**
for (group_number = 1, j=0; group_number <= 10; group_number++) {		
if (group_number == 1) {		
if(fap_group_mask[1] & 0x1)		
decode_viseme()		
if(fap_group_mask[1] & 0x2)		
decode_expression()		
} else {		
for (i= 0; i<NFAP[group_number]; i++, j++) {		
if(fap_group_mask[group_number] & (1 < < i)) {		
aa_decode(ifap_Q[j],ifap_cum_freq[j])		
}		
}		
}		
}		
}		

6.2.10.5 Decode pfap

decode_pfap(){	**No. of bits**	**Mnemonic**
for (group_number = 1, j=0; group_number <= 10; group_number++) {		
if (group_number == 1) {		
if(fap_group_mask[1] & 0x1)		
decode_viseme()		
if(fap_group_mask[1] & 0x2)		
decode_expression()		
} else {		
for (i= 0; i<NFAP[group_number]; i++, j++) {		

if(fap_group_mask[group_number] & (1 << i)) {		
aa_decode(pfap_diff[j], pfap_cum_freq[j])		
}		
}		
}		
}		
}		

6.2.10.6 Decode viseme and expression

decode_viseme() {	No. of bits	Mnemonic
aa_decode(viseme_select1Q, viseme_select1_cum_freq)		vlclbf
aa_decode(viseme_select2Q, viseme_select2_cum_freq)		vlclbf
aa_decode(viseme_blendQ, viseme_blend_cum_freq)		vlclbf
viseme_ def	1	bslbf
}		

decode_expression() {	No. of bits	Mnemonic
aa_decode(expression_select1Q, expression_select1_cum_freq) aa_decode(expression_intensity1Q, expression_intensity1_cum_freq)		vlclbf vlclbf
aa_decode(expression_select2Q, expression_select2_cum_freq)		vlclbf
aa_decode(expression_intensity2Q, expression_intensity2_cum_freq)		vlclbf
aa_decode(expression_blendQ, expression_blend_cum_freq)		vlclbf
init_face	1	bslbf
expression_def	1	bslbf
}		

6.2.10.7 Decode i_segment

decode_i_segment(){	No. of bits	Mnemonic
for (group_number= 1, j=0; group_number<= 10; group_number++) {		
if (group_number == 1) {		
if(fap_group_mask[1] & 0x1)		
decode_i_viseme_segment()		
if(fap_group_mask[1] & 0x2)		
decode_i_expression_segment()		
} else {		
for(i=0; i<NFAP[group_number]; i++, j++) {		
if(fap_group_mask[group_number] & (1 << i)) {		
decode_i_dc(dc_Q[j])		
decode_ac(ac_Q[j])		
}		
}		
}		

}		
}		

6.2.10.8 Decode p_segment

decode_p_segment(){	No. of bits	Mnemonic
for (group_number = 1, j =0; group_number <= 10; group_number++) {		

if (group_number == 1) {
if(fap_group_mask[1] & 0x1)
decode_p_viseme_segment()
if(fap_group_mask[1] & 0x2)
decode_p_expression_segment()
} else {		

for (i=0; i<NFAP[group_number]; i++, j++) {		
If(fap_group_mask[group_number] & (1 << i)) {		
decode_p_dc(dc_Q[j])		
decode_ac(ac_Q[j])		
}		
}		
}		
}		
}		

6.2.10.9 Decode viseme and expression

decode_i_viseme_segment(){	No. of bits	Mnemonic
viseme_segment_select1q[0]	4	uimsbf
viseme_segment_select2q[0]	4	uimsbf
viseme_segment_blendq[0]	6	uimsbf
viseme_segment_def[0]	1	bslbf
for (k=1; k<16, k++) {		
viseme_segment_select1q_diff[k]		vlclbf
viseme_segment_select2q_diff[k]		vlclbf
viseme_segment_blendq_diff[k]		vlclbf
viseme_segment_def[k]	1	bslbf
}		
}		

decode_p _viseme_segment(){	No. of bits	Mnemonic
for (k=0; k<16, k++) {		
viseme_segment_select1q_diff[k]		vlclbf
viseme_segment_select2q_diff[k]		vlclbf
viseme_segment_blendq_diff[k]		vlclbf
viseme_segment_def[k]	1	bslbf

}		
}		

decode_i_expression_segment(){	**No. of bits**	**Mnemonic**
expression_segment_select1q[0]	4	uimsbf
expression_segment_select2q[0]	4	uimsbf
expression_segment_intensity1q[0]	6	uimsbf
expression_segment_intensity2q[0]	6	uimsbf
expression_segment_init_face[0]	1	bslbf
expression_segment_def[0]	1	bslbf
for (k=1; k<16, k++) {		
expression_segment_select1q_diff[k]		vlclbf
expression_segment_select2q_diff[k]		vlclbf
expression_segment_intensity1q_diff[k]		vlclbf
expression_segment_ intensity2q_diff[k]		vlclbf
expression_segment_init_face[k]	1	bslbf
expression_segment_def[k]	1	bslbf
}		
}		

decode_p _expression_segment(){	**No. of bits**	**Mnemonic**
for (k=0; k<16, k++) {		
expression_segment_select1q_diff[k]		vlclbf
expression_segment_select2q_diff[k]		vlclbf
expression_segment_intensity1q_diff[k]		vlclbf
expression_segment_intensity2q_diff[k]		vlclbf
expression_segment_init_face[k]	1	bslbf
expression_segment_def[k]	1	bslbf
}		
}		

decode_i_dc(dc_q) {	**No. of bits**	**Mnemonic**
dc_q	16	simsbf
if(dc_q == -256*128)		
dc_q	31	simsbf
}		

decode_p_dc(dc_q_diff) {	**No. of bits**	**Mnemonic**
dc_q_diff		vlclbf
dc_q_diff = dc_q_diff- 256		
if(dc_q_diff == -256)		
dc_q_diff	16	simsbf
if(dc_Q == 0-256*128)		
dc_q_diff	32	simsbf
}		

decode_ac(ac_Q[i]) {	No. of bits	Mnemonic
this = 0		
next = 0		
while(next < 15) {		
count_of_runs		vlclbf
if (count_of_runs == 15)		
next = 16		
else {		
next = this+1+count_of_runs		
for (n=this+1; n<next; n++)		
ac_q[i][n] = 0		
ac_q[i][next]		vlclbf
if(ac_q[i][next] == 256)		
decode_i_dc(ac_q[i][next])		
else		
ac_q[i][next] = ac_q[i][next]-256		
this = next		
}		
}		
}		

6.2.10.10 Decode bap min max

decode_bap_new_minmax() {	No. of bits	Mnemonic
if (bap_is_i_new_max) {		
for (group_number=1;group_number<= BAP_NUM_GROUPS; group_number++)		
for (i=0; i < NBAP_GROUP[group_number]; i++) {		
j=BAPS_IN_GROUP[group_number][i]		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (bap_group_mask[group_number] & (1 <<i))		
bap_i_new_max[j]	5	uimsbf
}		
if (bap_is_i_new_min) {		
for (group_number = 1; group_number <= BAP_NUM_GROUPS; group_number++)		
for (i=0; i < NBAP_GROUP[group_number]; i++) {		
j=BAPS_IN_GROUP[group_number][i]		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (bap_group_mask[group_number] & (1 <<i))		
bap_i_new_min[j]	5	uimsbf
}		
if (bap_is_p_new_max) {		
for (group_number = 1; group_number <= BAP_NUM_GROUPS; group_number++)		
for (i=0; i < NBAP_GROUP[group_number]; i++) {		
j=BAPS_IN_GROUP[group_number][i]		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (bap_group_mask[group_number] & (1 <<i))		
bap_p_new_max[j]	5	uimsbf
}		
if (bap_is_p_new_min) {		
for (group_number = 1; group_number <= BAP_NUM_GROUPS; group_number++)		
for (i=0; i < NBAP_GROUP[group_number]; i++) {		
j=BAPS_IN_GROUP[group_number][i]		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (bap_group_mask[group_number] & (1 <<i))		
bap_p_new_min[j]	5	uimsbf
}		
}		
}		

6.2.10.11 Decode ibap

decode_ibap(){	No. of bits	Mnemonic
for (group_number = 1; group_number <= BAP_NUM_GROUPS; group_number++) {		
for (i= 0; i<NBAP_GROUP[group_number]; i++) {		
j=BAPS_IN_GROUP[group_number][i]		
if(bap_group_mask[group_number] & (1 << i)) {		
aa_decode(ibap_Q[j],ibap_cum_freq[j])		
}		
}		
}		
}		

6.2.10.12 Decode pbap

decode_pbap() {	No. of bits	Mnemonic
for (group_number = 1; group_number <= BAP_NUM_GROUPS; group_number++) {		
for (i= 0; i<NBAP_GROUP[group_number]; i++) {		
j=BAPS_IN_GROUP[group_number][i]		
if(bap_group_mask[group_number] & (1 << i)) { aa_decode(pbap_diff[j], pbap_cum_freq[j]) } } } }		

6.2.10.13 Decode bap i segment

decode_bap_i_segment(){	No. of bits	Mnemonic
for (group_number= 1; group_number<= BAP_NUM_GROUPS; group_number++) {		
for(i=0; i<NBAP_GROUP[group_number]; i++) {		
if(bap_group_mask[group_number] & (1 << i)) { j=BAPS_IN_GROUP[group_number][i] decode_i_dc(dc_Q[j]) decode_ac(ac_Q[j]) } }		
}		
}		

6.2.10.14 Decode bap p segment

decode_bap_p_segment(){	No. of bits	Mnemonic
for (group_number = 1; group_number <= BAP_NUM_GROUPS; group_number++) {		
for (i=0; i<NBAP_GROUP[group_number]; i++) {		
If(bap_group_mask[group_number] & (1 << i)) {		
j = BAPS_IN_GROUP[group_number][i]		
decode_p_dc(dc_Q[j])		
decode_ac(ac_Q[j])		
}		
}		
}		
}		

6.2.11 3D Mesh Object

6.2.11.1 3D_Mesh_Object

3D_Mesh_Object () {	No. of bits	Mnemonic
3D_MO_start_code	16	uimsbf
3D_Mesh_Object_Header()		
do {		
3D_Mesh_Object_Layer()		
} while (nextbits_bytealigned() == **3D_MOL_start_code**)		
}		

6.2.11.2 3D_Mesh_Object_Header

3D_Mesh_Object_Header() {	No. of bits	Mnemonic
ccw	1	bslbf
convex	1	bslbf
solid	1	bslbf
creaseAngle	6	**uimsbf**
coord_header()		
normal_header()		
color_header()		
texCoord_header()		
ce_SNHC	1	**bslbf**
if (**ce_SNHC** == '1')		
ce_SNHC _header()		
}		

6.2.11.3 3D_Mesh_Object_Layer

3D_Mesh_Object_Layer () {	No. of bits	Mnemonic
3D_MOL_start_code	16	uimsbf
mol_id	8	uimsbf
if (**ce_SNHC** == '1') {		
ce_SNHC _n_vertices	24	uimsbf
ce_SNHC _n_triangles	24	uimsbf
ce_SNHC _n_edges	24	uimsbf
}		
if (**mol_id** == 0)		
3DMeshObject_Base_Layer()		
else		
3DMeshObject_Refinement_Layer()		
}		

6.2.11.4 3D_Mesh_Object_Base_Layer

3DmeshObject_Base_Layer()	No. of bits	Mnemonic
do {		
3D_MOBL_start_code	16	uimsbf
mobl_id	8	uimsbf
while (!bytealigned())		
one_bit	1	bslbf
qf_start()		
if (**3D_MOBL_start_code** == "partition_type_0") {		
do {		
connected_component()		
qf_decode(**last_component**, last_component_context)		vlclbf
} while (**last_component** == '0')		
}		
else if (**3D_MOBL_start_code** == "partition_type_1") {		
vg_number=0		
do {		
vertex_graph()		
vg_number++		
qf_decode(**has_stitches**, has_stitches_context)		vlclbf
if (**has_stitches** == '1') stitches()		
qf_decode(**codap_last_vg**, codap_last_vg_context)		vlclbf
} while (**codap_last_vg** == '0')		
}		
else if (**3D_MOBL_start_code** == "partition_type_2") {		
if(vg_number > 1)		
qf_decode(**codap_vg_id**)		vlclbf
qf_decode(**codap_left_bloop_idx**)		vlclbf
qf_decode(**codap_right_bloop_idx**)		vlclbf
qf_decode(**codap_bdry_pred**)		vlclbf
triangle_tree()		
triangle_data()		
}		
} while (nextbits_bytealigned() == **3D_MOBL_start_code**)		
}		

6.2.11.5 coord_header

coord_h eader() {	No. of bits	Mnemonic
coord_binding	2	uimsbf
coord_bbox	1	bslbf
if (**coord_bbox** == '1') {		
coord_xmin	32	bslbf
coord_ymin	32	bslbf
coord_zmin	32	bslbf
coord_size	32	bslbf
}		
coord_quant	5	uimsbf
coord_pred_type	2	uimsbf
if (**coord_pred_type**=="tree_prediction" || **coord_pred_type**=='"parallelogram_prediction") {		
coord_nlambda	2	uimsbf
for (i=1; i<**coord_nlambda**; i++)		
coord_lambda	4-27	simsbf
}		
}		

6.2.11.6 normal_header

normal_header() {	No. of bits	Mnemonic
normal_binding	2	uimsbf
if (**normal_binding** != " not_bound") {		
normal_bbox	1	bslbf
normal_quant	5	uimsbf
normal_pred_type	2	uimsbf
if (**normal_pred_type** == " tree_prediction"|| **normal_pred_type** == "parallelogram_prediction") {		
normal_nlambda	2	uimsbf
for (i=1; i<**normal_nlambda**; i++)		
normal_lambda	3-17	simsbf
}		
}		
}		

6.2.11.7 color_header

color_header() {	No. of bits	Mnemonic
color_binding	2	uimsbf
if (color_binding != "not_bound") {		
color_bbox	1	bslbf
if (color_bbox == '1') {		
color_rmin	32	bslbf
color_gmin	32	bslbf
color_bmin	32	bslbf
color_size	32	bslbf
}		
color_quant	5	uimsbf
color_pred_type	2	uimsbf
if (color_pred_type=="tree_prediction" || color_pred_type=='"parallelogram_prediction") {		
color_nlambda	2	uimsbf
for (i=1; i<color_nlambda ; i++)		
color_lambda	4-19	simsbf
}		
}		
}		

6.2.11.8 texCoord_header

texCoord_header() {	No. of bits	Mnemonic
texCoord_binding	2	uimsbf
if (texCoord_binding != "not_bound") {		
texCoord_bbox	1	bslbf
if (texCoord_bbox == '1') {		
texCoord_umin	32	bslbf
texCoord_vmin	32	bslbf
texCoord_size	32	bslbf
}		
texCoord_quant	5	uimsbf
texCoord_pred_type	2	uimsbf
if (texCoord_pred_type=="tree_prediction" || texCoord_pred_type=="parallelogram_prediction") {		
texCoord_nlambda	2	uimsbf
for (i=1; i<texCoord_nlambda; i++)		
texCoord_lambda	4-19	simsbf
}		
}		
}		

6.2.11.9 ce_SNHC_header

ce_SNHC_header() {	No. of bits	Mnemonic
ce_SNHC_n_proj_surface_spheres	4	uimsbf
if (ce_SNHC_n_proj_surface_spheres != 0) {		
ce_SNHC_x_coord_center_point	32	bslbf
ce_SNHC_y_coord_center_point	32	bslbf
ce_SNHC_z_coord_center_point	32	bslbf
ce_SNHC_normalized_screen_distance_factor	8	uimsbf
for (i=0; i<ce_SNHC_n_proj_surface_spheres; i++) {		
ce_SNHC_radius	32	bslbf
ce_SNHC_min_proj_purface	32	bslbf
ce_SNHC_n_proj_points	8	uimsbf
for (j=0; j<ce_SNHC_n_proj_points; j++) {		
ce_SNHC_sphere_point_coord	11	uimsbf
ce_SNHC_proj_surface	32	bslbf
}		
}		
}		
}		

6.2.11.10 connected_component

connected_component() {	No. of bits	Mnemonic
vertex_graph()		
gf_decode(has_stitches, has_stitches_context)		vlclbf
if (has_stitches == '1')		
stitches()		
triangle_tree()		
triangle_da ta()		
}		

6.2.11.11 vertex_graph

vertex_graph() {	No. of bits	Mnemonic
gf_decode(**vg_simple**, vg_simple_context)		vlclbf
depth = 0		
code_last = '1'		
openloops = 0		
do {		
do {		
if (code_last == '1') {		
gf_decode(**vg_last**, vg_last_context)		vlclbf
if (openloops > 0) {		
gf_decode(**vg_forward_run**, vg_forward_run_context)		vlclbf
if (**vg_forward_run** == '0') {		
openloops--		
if (openloops > 0)		
gf_decode(**vg_loop_index**, vg_loop_index_context)		vlclbf
break		
}		
}		
}		
gf_decode(**vg_run_length**, vg_run_length_context)		vlclbf
gf_decode(**vg_leaf**, vg_leaf_context)		vlclbf
if (**vg_leaf** == '1' && **vg_simple** == '0') {		
gf_decode(**vg_loop**, vg_loop_context)		vlclbf
if (**vg_loop** == '1')		
openloops++		
}		
} while (0)		
if (**vg_leaf** == '1' && **vg_last** == '1' && code_last == '1')		
depth --		
if (**vg_leaf** == '0' && (**vg_last** == '0' || code_last == '0'))		
depth++		
code_last = **vg_leaf**		
} while (depth >= 0)		
}		

6.2.11.12 stitches

stitches() {	No. of bits	Mnemonic
for each vertex in connected_component {		
qf_decode(**stitch_cmd**, stitch_cmd_context)		vlclbf
if (**stitch_cmd**) {		
qf_decode(**stitch_pop_or_get**, stitch_pop_or_get_context)		vlclbf
if (**stitch_pop_or_get** == '1') {		
qf_decode(**stitch_pop**, stitch_pop_context)		vlclbf
qf_decode(**stitch_stack_index**, stitch_stack_index_context)		vlclbf
qf_decode(**stitch_incr_length**, stitch_incr_length_context)		vlclbf
if (**stitch_incr_length** != 0)		
qf_decode(**stitch_incr_length_sign**, stitch_incr_length_sign_context)		vlclbf
qf_decode(**stitch_push**, stitch_push_context)		vlclbf
if (total length >0)		
qf_decode(**stitch_reverse**, stitch_reverse_context)		vlclbf
}		
else		
qf_decode(**stitch_length**, stitch_length_context)		vlclbf
}		
}		
}		

6.2.11.13 triangle_tree

triangle_tree() {	No. of bits	Mnemonic
depth = 0		
ntriangles = 0		
branch_position = -2		
do {		
qf_decode(**tt_run_length**, tt_run_length_context)		vlclbf
ntriangles += **tt_run_length**		
qf_decode(**tt_leaf**, tt_leaf_context)		vlclbf
if (**tt_leaf** == '1') {		
depth --		
}		
else {		
branch_position = ntriangles		
depth++		
}		
} while (depth >= 0)		
if (**3D_MOBL_start_code** == "partition_type_2")		
if (**codap_right_bloop_idx** – **codap_left_bloop_idx** – 1 > ntriangles) {		
if (branch_position == ntriangles – 2) {		
qf_decode(**codap_branch_len**, codap_branch_len_context) ntriangles -= 2		vlclbf
}		
else ntriangles--		
}		
}		

6.2.11.14 triangle_data

triangle_data() {	No. of bits	Mnemonic
qf_decode(**triangulated**, triangulated_context)		vlclbf
depth=0		
root_triangle()		
for (i=1; i<ntriangles; i++)		
triangle()		
}		

6.2.11.15 root_triangle

root_triangle() {	No. of bits	Mnemonic
if (marching_triangle)		
qf_decode(**marching_pattern**, marching_pattern_context[**marching_pattern**])		vlclbf
else {		
if (**3D_MOBL_start_code** == "partition_type_2")		
if (**tt_leaf** == '0' && depth==0)		
qf_decode(**td_orientation**, td_orientation_context)		vlclbf
if (**tt_leaf** == '0')		
depth++		
else		
depth --		
}		
if (**3D_MOBL_start_code** == "partition_type_2")		
if (**triangulated** == '0')		
qf_decode(**polygon_edge**, polygon_edge_context[**polygon_edge**])		vlclbf
root_coord()		
root_normal()		
root_color()		
root_texCoord()		
}		

root_coord() {	No. of bits	Mnemonic
if (**3D_MOBL_start_code** == "partition_type_2") {		
if (visited[vertex_index] == 0) {		
root_coord_sample()		
if (visited[vertex_index] == 0) {		

coord_sample() coord_sample()		
}		
}		
}		
else {		
root_coord_sample()		
coord_sample()		
coord_sample()		
}		
}		

root_normal() {	**No. of bits**	**Mnemonic**
if (**normal_binding** != “not_bound”)		
if (**3D_MOBL_start_code** == “partition_type_2”) {		
if (**normal_binding** != “bound_per_vertex” || visited[vertex_index] == 0) {		
root_normal_sample()		
if (normal_binding != “ bound_per_face” && (normal_binding != “bound_per_vertex” || visited[v ertex_index] == 0)) {		
normal_sample()		
normal_sample()		
}		
}		
else {		
root_normal_sample()		
if (**normal_binding** != “bound_per_face”) {		
normal_sample()		
normal_sample()		
}		
}		
}		

root_color() {	**No. of bits**	**Mnemonic**

if (**color_binding** != “not_bound”)
if (**3D_MOBL_start_code** == “partition_type_2”) {
if (**color_binding** != “bound_per_vertex” || visited[vertex_index] == 0) {		

root_color_sample() **if (color_binding != “ bound_per_face” && (color_binding != “bound_per_vertex” ||visited[vertex_index] == 0)) {**		

color_sample()		
color_sample()		
}		
}		
else {		
root_color_sample()		
if (**color_binding** != "bound_per_face") {		
color_sample()		
color_sample()		
}		
}		
}		

root_texCoord() {	**No. of bits**	**Mnemonic**

if (**texCoord_binding** != " not_bound")
if (**3D_MOBL_start_code** == "partition_type_2") {
if (**texCoord_binding** != "bound_per_vertex" || visited[vertex_index] == 0) {		

root_texCoord_sample()		
if (texCoord_binding!= "bound_per_vertex"|| visited[vertex_index] == 0) {		
texCoord_sample()		
texCoord_sample()		
}		
}		
else {		
root_texCoord_sample() texCoord_sample() texCoord_sample() }		
}		

6.2.11.16 triangle

triangle() {	No. of bits	Mnemonic
if (marching_triangle)		
gf_decode(**marching_edge**, marching_ edge_context[**marching_edge**])		vlclbf
else {		
if (**3D_MOBL_start_code** == “partition_type_2”)		
if (**tt_leaf** == ‘0’ && depth==0)		
gf_decode(**td_orientation**, td_orientation_context)		vlclbf
if (**tt_leaf** == ‘0’)		
depth++		
else		
depth –		
if (**triangulated** == ‘0’)		
gf_decode(**polygon_edge**, polygon_edge_context[**polygon_edge**])		vlclbf
coord()		
normal()		
color()		
texCoord()		
}		

coord() {	No. of bits	Mnemonic
if (**3D_MOBL_start_code** == “partition_type_2”) {		
if (visited[vertex_index] == 0)		
if (no_ancestors)		
root_coord_sample()		
else		
coord_sample()		
}		
else {		
if (visited[vertex_index] == 0)		
coord_sample()		
}		
}		

normal() {	No. of bits	Mnemonic
if (**normal_binding** == “bound_per_vertex”) {		
if (**3D_MOBL_start_code** == “partition_type_2”) {		
if (visited[vertex_index] == 0)		
if (no_ancestors)		
root_normal_sample()		
else		
normal_sample()		
}		

else {		
if (visited[vertex_index] == 0)		
normal_sample()		
}		
} else if (**normal_binding** == "bound_per_face") {		
if (**triangulated** == '1' || **polygon_edge** == '1')		
normal_sample()		
} else if (**normal_binding** == "bound_per_corner") {		
if (**triangulated** == '1' || **polygon_edge** == '1') {		
normal_sample()		
normal_sample()		
}		
normal_sample()		
}		
}		

color() {	**No. of bits**	**Mnemonic**
if (**color_binding** == "bound_per_vertex") {		
if (**3D_MOBL_start_code** == "partition_type_2") {		
if (visited[vertex_index] == 0)		
if (no_ancestors)		
root_color_sample()		
else		
color_sample()		
}		
else {		
if (visited[vertex_index] == 0)		
color_sample()		
}		
} else if (**color_binding** == "bound_per_face") {		
if (**triangulated** == '1' || **polygon_edge** == '1')		
color_sample()		
} else if (**color_binding** == "bound_per_corner") {		
if (**triangulated** == '1' || **polygon_edge** == '1') {		
color_sample()		
color_sample()		
}		
color_sample()		
}		
}		

texCoord() {	**No. of bits**	**Mnemonic**
if (**texCoord_binding** == "bound_per_vertex") {		
if (**3D_MOBL_start_code** == "partition_type_2") {		
if (visited[vertex_index] == 0)		

if (no_ancestors)		
root_texCoord_sample()		
else		
texCoord_sample()		
}		
else {		
if (visited[vertex_index] == 0)		
texCoord_sample()		
}		
} else if (**texCoord_binding** == "bound_per_corner") {		
if (**triangulated** == '1' || **polygon_edge** == '1') {		
texCoord_sample()		
texCoord_sample()		
}		
texCoord_sample()		
}		
}		

root_coord_sample() {	**No. of bits**	**Mnemonic**
for (i=0; i<3; i++)		
for (j=0; j<**coord_quant**; j++)		
qf_decode(**coord_bit**, zero_context)		vlclbf
}		

root_normal_sample() {	**No. of bits**	**Mnemonic**
for (i=0; i<1; i++)		
for (j=0; j<**normal_quant**; j++)		
qf_decode(**normal_bit**, zero_context)		vlclbf
}		

root_color_sample() {	**No. of bits**	**Mnemonic**
for (i=0; i<3; i++)		
for (j=0; j<**color_quant**; j++)		
qf_decode(**color_bit**, zero_context)		vlclbf
}		

root_texCoord_sample() {	**No. of bits**	**Mnemonic**
for (i=0; i<2; i++)		
for (j=0; j<**texCoord_quant**; j++)		
qf_decode(**texCoord_bit**, zero_context)		vlclbf
}		

coord_sample() {	**No. of bits**	**Mnemonic**
for (i=0; i<3; i++) {		

j=0		
do {		
qf_decode(**coord_leading_bit**, coord_leading_bit_context[3*j+i])		vlclbf
j++		
} while (j<**coord_quant** && **coord_leading_bit** == '0')		
if (**coord_leading_bit** == '1') {		
qf_decode(**coord_sign_bit**, zero_context)		vlclbf
do {		
qf_decode(**coord_trailing_bit**, zero_context)		vlclbf
} while (j<**coord_quant**)		
}		
}		
}		

normal_sample() {	**No. of bits**	**Mnemonic**
for (i=0; i<1; i++) {		
j=0		
do {		
qf_decode(**normal_leading_bit**, normal_leading_bit_context[j])		vlclbf
j++		
} while (j<**normal_quant** && **normal_leading_bit** == '0')		
if (**normal_leading_bit** == ' 1'){		
qf_decode(**normal_sign_bit**, zero_context)		vlclbf
do {		
qf_decode(**normal_trailing_bit**, zero_context)		vlclbf
} while (j<**normal_quant**)		
}		
}		
}		

color_sample() {	**No. of bits**	**Mnemonic**
for (i=0; i<3; i++) {		
j=0		
do {		
qf_decode(**color_leading_bit**, color_leading_bit_context[3*j+i])		vlclbf

j++
} while (j<**color_quant** && **color_leading_bit** == '0')
if (**color_leading_bit** == '1') {
qf_decode(**color_sign_bit**, zero_context)
do {
qf_decode(**color_trailing_bit**, zero_context)
		
		vlclbf
		
		vlclbf

} while (j<**color_quant**)		
}		
}		
}		

texCoord_sample() {	No. of bits	Mnemonic
for (i=0; i<2; i++) {		
j=0		
do {		
gf_decode(**texCoord_leading_bit**, texCoord_leading_bit_context[2*j+i])		vlclbf
j++		
} while (j<**texCoord_quant** && **texCoord_leading_bit** == '0')		
if (**texCoord_leading_bit** == '1') {		
gf_decode(**texCoord_sign_bit**, zero_context)		vlclbf
do {		
gf_decode(**texCoord_trailing_bit**, zero_context)		vlclbf
} while (j<**texCoord_quant**)		
}		
}		
}		

6.2.11.17 3DMeshObject_Refinement_Layer

3DMeshObject_Refinement_Layer () {	No. of bits	Mnemonic
do {		
3D_MORL_start_code	16	uimsbf
morl_id	8	uimsbf
connectivity_update	2	uimsbf
pre_smoothing	1	bslbf
if(**pre_smoothing** == '1')		
pre_smoothing_parameters()		
post_smoothing	1	bslbf
if(**post_smoothing** == '1')		
post_smoothing_parameters()		
while (!bytealigned())		
one_bit	1	bslbf
qf_start()		
if(**connectivity_update** == "fs_update")		
fs_pre_update()		

if(**pre_smoothing** == '1' || **post_smoothing** == '1')
smoothing_constraints()
/* apply pre smoothing step */
if(**connectivity_update** == "fs_update")
fs_post_update()
if(**connectivity_update** != " not_updated")		

qf_decode(**other_update**, zero_context)		vlclbf
if(**connectivity_update** =="not_updated" || **other_update** == '1')		
other_property_update()		
/* apply post smoothing step */		
} while (nextbits_bytealigned() == **3D_MORL_start_code**)		
}		

6.2.11.17.1 pre_smoothing_parameters

pre_smoothing_parameters() {	No. of bits	Mnemonic
pre_smoothing_n	8	uimsbf
pre_smoothing_lambda	32	bslbf
pre_smoothing_mu	32	bslbf
}		

6.2.11.17.2 post_smoothing_parameters

post_smoothing_parameters() {	No. of bits	Mnemonic
post_smoothing_n	8	uimsbf
post_smoothing_lambda	32	bslbf
post_smoothing_mu	32	bslbf
}		

6.2.11.17.3 fs_pre_update

fs_pre_update() {	No. of bits	Mnemonic
for each connected component {		
forest()		
for each tree in forest {		
triangle_tree() /* for each tree loop vertex set visited[vertex_index]='1' */ triangle_data()		
}		
}		
}		

forest () {	No. of bits	Mnemonic
for each edge in connected component		
if (creates no loop in forest)		
qf_decode(**pfs_forest_edge**, pfs_forest_edge_context)		vlclbf
}		

6.2.11.17.4 smoothing_constraints

smoothing_constraints() {	No. of bits	Mnemonic
qf_decode(**smooth_with_sharp_edges**, zero_context)		vlclbf
if (**smooth_with_sharp_edges** == '1')		
sharp_edge_marks()		
qf_decode(**smooth_with_fixed_vertices** , zero_context)		vlclbf
if (**smooth_with_fixed_vertices** == '1')		
fixed_vertex_marks()		
}		

sharp_edge_marks () {	No. of bits	Mnemonic
for each edge		
qf_decode(**smooth_sharp_edge** , smooth_sharp_edge_context)		vlclbf
}		

fixed_vertex_marks () {	No. of bits	Mnemonic
for each vertex		
qf_decode(**smooth_fixed_vertex**, smooth_fixed_vertex_context)		vlclbf
}		

6.2.11.17.5 fs_post_update

fs_post_update() {	No. of bits	Mnemonic
for each connected component {		
for each tree in forest		
tree_loop_property_update()		
}		

tree_loop_property_up date () {	No. of bits	Mnemonic
loop_coord_update()		
loop_normal_update()		
loop_color_update()		
loop_texCoord_update()		
}		

loop_coord_update () {	No. of bits	Mnemonic
for each tree loop vertex		
coord_sample()		
}		

loop_normal_update () {	No. of bits	Mnemonic
if (**normal_binding** == “bound_per_vertex”) {		
for each tree loop vertex		
normal_sample()		
}		
else if (**normal_binding** == “bound_per_face”) {		
for each tree loop face		
normal_sample()		
}		
else if (**normal_binding** == “bound_per_corner”) {		
for each tree loop corner		
normal_sample()		
}		
}		

loop_color_update () {	No. of bits	Mnemonic
if (**color_binding** == “ bound_per_vertex”) {		
for each tree loop vertex		
color_sample()		
}		
else if (**color_binding** == “ bound_per_face”) {		
for each tree loop face		
color_sample()		
}		
else if (**color_binding** == “ bound_per_corner”) {		

for each tree loop corner		
color_sample()		
}		
}		

loop_texCoord_update () {	No. of bits	Mnemonic
if (**texCoord_binding** == “bound_per_vertex”) {		
for each tree loop vertex		
texCoord_sample()		
}		
else if (**texCoord_binding** == “bound_per_corner”) {		
for each tree loop corner		
texCoord_sample()		
}		
}		

6.2.11.17.6 other_property_update

other_property_update() {	No. of bits	Mnemonic
other_coord_update()		
other_normal_update()		
other_color_update()		
other_texCoord_update()		
}		

other_coord_update () {	No. of bits	Mnemonic
for each vertex in mesh		
if (vertex is not a tree loop vertex)		
coord_sample()		
}		

other_normal_update () {	No. of bits	Mnemonic
if (**normal_binding** == "bound_per_vertex") {		
for each vertex in mesh		

if (vertex is not a tree loop vertex)
normal_sample()
}		

else if (**normal_binding** == "bound_per_face") {
for each face in mesh
if (face is not a tree loop face)		

normal_sample()		
}		
else if (**normal_binding** == "bound_per_corner") {		
for each corner in mesh		
if (corner is not a tree loop corner)		
normal_sample()		
}		
}		

other_color_update () {	No. of bits	Mnemonic
if (**color_binding** == " bound_per_vertex") {		
for each vertex in mesh		
if (vertex is not a tree loop vertex)		
color_sample()		
}		
else if (**color_binding** == " bound_per_face") {		
for each face in mesh		

if (face is not a tree loop face)
color_sample()
}		

else if (**color_binding** == " bound_per_corner") { for each corner in mesh		

if (corner is not a tree loop corne r)		
color_sample()		
}		
}		

other_texCoord_update () {	No. of bits	Mnemonic
if (**texCoord_binding** == “bound_per_vertex”) {		
for each vertex in tree loop		
if (vertex is not a tree loop vertex)		
texCoord_sample()		
}		
else if (**texCoord_binding** == “bound_per_corner”) {		
for each corner in mesh		
if (corner is not a tree loop corner)		
texCoord_sample()		
}		
}		

6.2.12 Upstream message

6.2.12.1 upstream_message

upstream_message() {	No. of bits	Mnemonic
upstream_message_type	3	bslbf
If (upstream_message_type == Video_NEWPRED)		

upstream_Video_NEWPRED()
if (upstream_message_type == SNHC_QoS)
upstream_SNHC_QoS()
byte_align_for_upstream()
}		

6.2.12.2 upstream_Video_NEWPRED

upstream_Video_NEWPRED() {	No. of bits	Mnemonic
newpred_upstream_message_type	2	bslbf
if (newpred_upstream_message_type == " NP_NACK")		
unreliable_flag	1	bslbf
vop_id	4-15	uimsbf
macroblock_number	1-14	uimsbf
if (newpred_upstream_message_type == " Intra Refresh Command")		
end_macroblock_number	1-14	uimsbf
if (newpred_upstream_message_type == " NP_NACK")		
requested_vop_id_for_prediction	4-15	uimsbf
}		

6.2.12.3 upstream_SNHC_QoS

upstream_SNHC_QoS() {	No. of bits	Mnemonic
screen_width	12	uimsbf
screen_height	12	uimsbf
n_rendering_modes	4	uimsbf
for (rendering_mode = 0 ; rendering_mode < n_rendering_modes; rendering_mode++) {		
rendering_mode_type	4	uimsbf
n_curves	8	uimsbf
for (j = 0 ; j < n_curves ; j++) {		
triangle_parameter	24	uimsbf
n_points_on_curve	8	uimsbf
for (j = 0 ; j < n_points_on_curve ; j++) {		
screen_coverage_parameter	8	uimsbf
frame_rate_value	12	bslbf
}		
}		
}		
}		

6.3 Visual bitstream semantics

6.3.1 Semantic rules for higher syntactic structures

This subclause details the rules that govern the way in which the higher level syntactic elements may be combined together to produce a legal bitstream. Subsequent subclauses detail the semantic meaning of all fields in the video bitstream.

6.3.2 Visual Object Sequence and Visual Object

visual_object_sequence_start_code: The visual_object_sequence_start_code is the bit string '000001B0' in hexadecimal. It initiates a visual session.

profile_and_level_indication: This is an 8-bit integer used to signal the profile and level identification. The meaning of the bits is given in Table G-1.

visual_object_sequence_end_code: The visual_object_sequence_end_code is the bit string '000001B1' in hexadecimal. It terminates a visual session.

visual_object_start_code: The visual_object_start_code is the bit string '000001B5' in hexadecimal. It initiates a visual object.

is_visual_object_identifier: This is a 1-bit code which when set to '1' indicates that version identification and priority is specified for the visual object. When set to '0', no version identification or priority needs to be specified.

visual_object_verid: This is a 4-bit code which identifies the version number of the visual object. Its meaning is defined in Table 6-4. When this field does not exist, the value of visual_object_verid is '0001'.

Table 6-4 -- Meaning of visual_object_verid

visual_object_verid	Meaning
0000	reserved
0001	ISO/IEC 14496-2:1999
0010	ISO/IEC 14496-2:1999/Amd. 1
0011 - 1111	reserved

visual_object_priority: This is a 3-bit code which specifies the priority of the visual object. It takes values between 1 and 7, with 1 representing the highest priority and 7, the lowest priority. The value of zero is reserved.

visual_object_type: The visual_object_type is a 4-bit code given in Table 6-5 which identifies the type of the visual object.

Table 6-5 -- Meaning of visual object type

code	visual object type
0000	reserved
0001	video ID
0010	still texture ID
0011	mesh ID
0100	FBA ID
0101	3D mesh ID
01101	reserved
:	:
:	:
1111	reserved

video_object_start_code: The video_object_start_code is a string of 32 bits. The first 27 bits are '0000 0000 0000 0000 0000 0001 000' in binary and the last 5-bits represent one of the values in the range of '00000' to '11111' in binary. The video_object_start_code marks a new video object.

video_object_id: This is given by the last 5-bits of the video_object_start_code. The video_object_id uniquely identifies a video object.

video_signal_type: A flag which if set to '1' indicates the presence of video_signal_type information.

video_format: This is a three bit integer indicating the representation of the pictures before being coded in accordance with this part of ISO/IEC 14496. Its meaning is defined in Table 6-6. If the video_signal_type() is not present in the bitstream then the video format may be assumed to be "Unspecified video format".

Table 6-6 -- Meaning of video_format

video_format	Meaning
000	Component
001	PAL
010	NTSC
011	SECAM
100	MAC
101	Unspecified video format
110	Reserved
111	Reserved

video_range: This one-bit flag indicates the black level and range of the luminance and chrominance signals.

colour_description: A flag which if set to '1' indicates the presence of colour_primaries, transfer_characteristics and matrix_coefficients in the bitstream.

colour_primaries: This 8-bit integer describes the chromaticity coordinates of the source primaries, and is defined in Table 6-7.

Table 6-7 -- Colour Primaries

Value	Primaries
0	(forbidden)
1	Recommendation ITU-R BT.709 primary x y green 0,300 0,600 blue 0,150 0,060 red 0,640 0,330 white D65 0,3127 0,3290
2	Unspecified Video Image characteristics are unknown.
3	Reserved
4	Recommendation ITU-R BT.470-2 System M primary x y green 0,210,71 blue 0,140,08 red 0,670,33 white C 0,310 0,316
5	Recommendation ITU-R BT.470-2 System B, G primary x y green 0,290,60 blue 0,150,06 red 0,640,33 white D65 0,3127 0,3290

6	SMPT E 170M primary x y green 0,310 0,595 blue 0,155 0,070 red 0,630 0,340 white D65 0,3127 0,3290
7	SMPTE 240M (1987) primary x y green 0,310 0,595 blue 0,155 0,070 red 0,630 0,340 white D65 0,3127 0,3290
8	Generic film (colour filters using Illuminant C) primary x y green 0,243 0,692 (Wratten 58) blue 0,145 0,049 (Wratten 47) red 0,681 0,319 (Wratten 25)
9-255	Reserved

In the case that video_signal_type() is not present in the bitstream or colour_description is zero the chromaticity is assumed to be that corresponding to colour_primaries having the value 1.

transfer_characteristics: This 8-bit integer describes the opto-electronic transfer characteristic of the source picture, and is defined in Table 6-8.

Table 6-8 -- Transfer Characteristics

Value	Transfer Characteristic
0	(forbidden)
1	Recommendation ITU-R BT.709 $V = 1{,}099\ L_C^{0{,}45} - 0{,}099$ for $1 \geq L_C \geq 0{,}018$ $V = 4{,}500\ L_C$ for $0{,}018 > L_C \geq 0$
2	Unspecified Video Image characteristics are unknown.
3	reserved
4	Recommendation ITU-R BT.470-2 System M Assumed display gamma 2,2
5	Recommendation ITU-R BT.470-2 System B, G Assumed display gamma 2,8
6	SMPTE 170M $V = 1{,}099\ L_C^{0{,}45} - 0{,}099$ for $1 \geq L_C \geq 0{,}018$ $V = 4{,}500\ L_C$ for $0{,}018 > L_C \geq 0$

7	SMPTE 240M (1987) $V = 1{,}1115\ L_c^{0{,}45} - 0{,}1115$ for $L_c \geq 0{,}0228$ $V = 4{,}0\ L_c$ for $0{,}0228 > L_c$
8	Linear transfer characteristics i.e. $V = L_c$
9	Logarithmic transfer characteristic (100:1 range) $V = 1.0 - \text{Log}_{10}(Lc)/2$ for $1 = L_c = 0.01$ $V = 0.0$ for $0.01 > L_c$
10	Logarithmic transfer characteristic (316.22777:1 range) $V = 1.0 - \text{Log}_{10}(Lc)/2.5$ for $1 = L_c = 0.0031622777$ $V = 0.0$ for $0.0031622777 > L_c$
11-255	reserved

In the case that video_signal_type() is not present in the bitstream or colour_description is zero the transfer characteristics are assumed to be those corresponding to transfer_characteristics having the value 1.

matrix_coefficients: This 8-bit integer describes the matrix coefficients used in deriving luminance and chrominance signals from the green, blue, and red primaries, and is defined in Table 6-9.

In this table:

E'_Y is analogue with values between 0 and 1

E'_{PB} and E'_{PR} are analogue between the values -0,5 and 0,5

E'_R, E'_G and E'_B are analogue with values between 0 and 1

White is defined as $E'_Y=1$, $E'_{PB}=0$, $E'_{PR}=0$; $E'_R = E'_G = E'_B = 1$.

Y, Cb and Cr are related to E'_Y, E'_{PB} and E'_{PR} by the following formulae:

if **video_range=0:**

$Y = (219 * 2^{n-8} * E'_Y) + 2^{n-4}$.

$Cb = (224 * 2^{n-8} * E'_{PB}) + 2^{n-1}$

$Cr = (224 * 2^{n-8} * E'_{PR}) + 2^{n-1}$

if **video_range=1:**

$Y = ((2^n - 1) * E'_Y)$

$Cb = ((2^n - 1) * E'_{PB}) + 2^{n-1}$

$Cr = ((2^n - 1) * E'_{PR}) + 2^{n-1}$

for n bit video.

For example, for 8 bit video,

video_range=0 gives a range of Y from 16 to 235, Cb and Cr from 16 to 240;

video_range=1 gives a range of Y from 0 to 255, Cb and Cr from 0 to 255.

Table 6-9 -- Matrix Coefficients

Value	Matrix
0	(forbidden)
1	Recommendation ITU-R BT.709 $E'_Y = 0,7152\ E'_G + 0,0722\ E'_B + 0,2126\ E'_R$ $E'_{PB} = -0,386\ E'_G + 0,500\ E'_B - 0,115\ E'_R$ $E'_{PR} = -0,454\ E'_G - 0,046\ E'_B + 0,500\ E'_R$
2	Unspecified Video Image characteristics are unknown.
3	reserved
4	FCC $E'_Y = 0,59\ E'_G + 0,11\ E'_B + 0,30\ E'_R$ $E'_{PB} = -0,331\ E'_G + 0,500\ E'_B - 0,169\ E'_R$ $E'_{PR} = -0,421\ E'_G - 0,079\ E'_B + 0,500\ E'_R$
5	Recommendation ITU-R BT.470-2 System B, G $E'_Y = 0,587\ E'_G + 0,114\ E'_B + 0,299\ E'_R$ $E'_{PB} = -0,331\ E'_G + 0,500\ E'_B - 0,169\ E'_R$ $E'_{PR} = -0,419\ E'_G - 0,081\ E'_B + 0,500\ E'_R$
6	SMPTE 170M $E'_Y = 0,587\ E'_G + 0,114\ E'_B + 0,299\ E'_R$ $E'_{PB} = -0,331\ E'_G + 0,500\ E'_B - 0,169\ E'_R$ $E'_{PR} = -0,419\ E'_G - 0,081\ E'_B + 0,500\ E'_R$
7	SMPTE 240M (1987) $E'_Y = 0,701\ E'_G + 0,087\ E'_B + 0,212\ E'_R$ $E'_{PB} = -0,384\ E'_G + 0,500\ E'_B - 0,116\ E'_R$ $E'_{PR} = -0,445\ E'_G - 0,055\ E'_B + 0,500\ E'_R$
8-255	reserved

In the case that video_signal_type() is not present in the bitstream or colour_description is zero the matrix coefficients are assumed to be those corresponding to matrix_coefficients having the value 1.

In the case that video_signal_type() is not present in the bitstream, video_range is assumed to have the value 0 (a range of Y from 16 to 235 for 8-bit video).

6.3.2.1 User data

user_data_start_code: The user_data_start_code is the bit string '000001B2' in hexadecimal. It identifies the beginning of user data. The user data continues until receipt of another start code.

user_data: This is an 8 bit integer, an arbitrary number of which may follow one another. User data is defined by users for their specific applications. In the series of consecutive user_data bytes there shall not be a string of 23 or more consecutive zero bits.

6.3.3 Video Object Layer

video_object_layer_start_code: The video_object_layer_start_code is a string of 32 bits. The first 28 bits are '0000 0000 0000 0000 0000 0001 0010' in binary and the last 4-bits represent one of the values in the range of '0000' to ' 1111'in binary. The video_object_layer_start_code marks a new video object layer.

video_object_layer_id: This is given by the last 4-bits of the video_object_layer_start_code. The video_object_layer_id uniquely identifies a video object layer.

short_video_header: The short_video_header is an internal flag which is set to 1 when an abbreviated header format is used for video content. This indicates video data which begins with a short_video_start_marker rather than a longer start code such as visual_object_ start_code. The short header format is included herein to provide forward compatibility with video codecs designed using the earlier video coding specification ITU-T Recommendation H.263. All decoders which support video objects shall support both header formats (short_video_header equal to 0 or 1) for the subset of video tools that is expressible in either form.

video_plane_with_short_header(): This is a syntax layer encapsulating a video plane which has only the limited set of capabilities available using the short header format.

random_accessible_vol: This flag may be set to "1" to indicate that every VOP in this VOL is individually decodable. If all of the VOPs in this VOL are intra-coded VOPs and some more conditions are satisfied then random_accessible_vol may be set to "1". The flag random_accessible_vol is not used by the decoding process. random_accessible_vol is intended to aid random access or editing capability. This shall be set to "0" if any of the VOPs in the VOL are non-intra coded or certain other conditions are not fulfilled.

video_object_type_indication: Constrains the following bitstream to use tools from the indicated object type only, e.g. Simple Object or Core Object, as shown in Table 6-10.

Table 6-10 -- FLC table for video_object_type indication

Video Object Type	Code
Reserved	00000000
Simple Object Type	00000001
Simple Scalable Object Type	00000010
Core Object Type	00000011
Main Object Type	00000100
N-bit Object Type	00000101
Basic Anim. 2D Texture	00000110
Anim. 2D Mesh	00000111
Simple Face	00001000
Still Scalable Texture	00001001
Advanced Real Time Simple	00001010
Core Scalable	00001011
Advanced Coding Efficiency	00001100
Advanced Scalable Texture	00001101
Simple FBA	00001110
Reserved	00001111 - 11111111

is_object_layer_identfier: This is a 1-bit code which when set to '1' indicates that version identification and priority is specified for the visual object layer. When set to '0', no version identification or priority needs to be specified.

video_object_layer_verid: This is a 4-bit code which identifies the version number of the video object layer. Its meaning is defined in Table 6-11. If both visual_object_verid and video_object_layer_verid exist, the semantics of

video_object_layer_verid supersedes the other. When this field does not exist, the value of video_object_layer_verid is substituted by the value of visual_object_verid.

Table 6-11 -- Meaning of video_object_layer_verid

video_object_layer_verid	Meaning
0000	Reserved
0001	ISO/IEC 14496-2:1999
0010	ISO/IEC 14496-2:1999/Amd. 1
0011 - 1111	Reserved

video_object_layer_priority: This is a 3-bit code which specifies the priority of the video object layer. It takes values between 1 and 7, with 1 representing the highest priority and 7, the lowest priority. The value of zero is reserved.

aspect_ratio_info: This is a four-bit integer which defines the value of pixel aspect ratio. Table 6-12 shows the meaning of the code. If aspect_ratio_info indicates extended PAR, pixel_aspect_ratio is represented by par_width and par_height. The par_width and par_height shall be relatively prime.

Table 6-12 -- Meaning of pixel aspect ratio

aspect_ratio_info	pixel aspect ratios
0000	Forbidden
0001	1:1 (Square)
0010	12:11 (625-type for 4:3 picture)
0011	10:11 (525-type for 4:3 picture)
0100	16:11 (625-type stretched for 16:9 picture)
0101	40:33 (525-type stretched for 16:9 picture)
0110-1110	Reserved
1111	extended PAR

par_width: This is an 8-bit unsigned integer which indicates the horizontal size of pixel aspect ratio. A zero value is forbidden.

par_height: This is an 8-bit unsigned integer which indicates the vertical size of pixel aspect ratio. A zero value is forbidden.

vol_control_parameters: This a one-bit flag which when set to '1' indicates presence of the following parameters: chroma_format, low_delay, and vbv_parameters.

chroma_format This is a two bit integer indicating the chrominance format as defined in the Table 6-13.

Table 6-13 -- Meaning of chroma_format

chroma_format	Meaning
00	reserved
01	4:2:0
10	reserved

11	reserved

low_delay : This is a one-bit flag which when set to '1' indicates the VOL contains no B-VOPs. If this flag is not present in the bitstream, the default value is 0 for visual object types that support B-VOP otherwise it is 1.

vbv_parameters: This is a one-bit flag which when set to '1' indicates presence of following VBV parameters: first_half_bit_rate, latter_half_bit_rate, first_half_vbv_buffer_size, latter_half_vbv_buffer_size, first_half_vbv_occupancy and latter_half_vbv_occupancy. The VBV constraint is defined in annex D.

first_half_bit_rate, latter_half_bit_rate: The bit rate is a 30-bit unsigned integer which specifies the bitrate of the bitstream measured in units of 400 bits/second, rounded upwards. The value zero is forbidden. This value is divided to two parts. The most significant bits are in first_half_bit_rate (15 bits) and the least significant bits are in latter_half_bit_rate (15 bits). The marker_bit is inserted between the first_half_bit_rate and the latter_half_bit_rate in order to avoid the resync_marker emulation. The instantaneous video object layer channel bit rate seen by the encoder is denoted by $R_{vol}(t)$ in bits per second. If the bit_rate (i.e. first_half_bit_rate and latter_half_bit_rate) field in the VOL header is present, it defines a peak rate (in units of 400 bits per second; a value of 0 is forbidden) such that $R_{vol}(t) <= 400 \times bit\_rate$ Note that $R_{vol}(t)$ counts only visual syntax for the current elementary stream (also see annex D).

first_half_vbv_buffer_size, latter_half_vbv_buffer_size: vbv_buffer_size is an 18-bit unsigned integer. This value is divided into two parts. The most significant bits are in first_half_vbv_buffer_size (15 bits) and the least significant bits are in latter_half_vbv_buffer_size (3 bits), The VBV buffer size is specified in units of 16384 bits. The value 0 for vbv_buffer_size is forbidden. Define $B = 16384 \times$ vbv_buffer_size to be the VBV buffer size in bits.

first_half_vbv_occupancy, latter_half_vbv_occupancy: The vbv_occupancy is a 26-bit unsigned integer. This value is divided to two parts. The most significant bits are in first_half_vbv_occupancy (11 bits) and the least significant bits are in latter_half_vbv_occupancy (15 bits). The marker_bit is inserted between the first_half_vbv_occupancy and the latter_half_vbv_occupancy in order to avoid the resync_marker emulation. The value of this integer is the VBV occupancy in 64-bit units just before the removal of the first VOP following the VOL header. The purpose for the quantity is to provide the initial condition for VBV buffer fullness.

video_object_layer_shape: This is a 2-bit integer defined in Table 6-14. It identifies the shape type of a video object layer.

Table 6-14 -- Video Object Layer shape type

Shape format	Meaning
00	rectangular
01	binary
10	binary only
11	grayscale

video_object_layer_shape_extension: This is a 4-bit integer defined in Table V2 - 1. It identifies the number (up to 3) and type of auxiliary components that can be used, including the grayscale shape (ALPHA) component. Only a limited number of types and combinations are defined in Table V2 - 1. More applications are possible by selection of the USER DEFINED type.

Table V2 - 1 -- Semantic meaning of video_object_layer_shape_extension

video_object_layer_shape_extension	aux_comp_type[0]	aux_comp_type[1]	aux_comp_type[2]	aux_comp_count

0000	ALPHA	NO	NO	1
0001	DISPARITY	NO	NO	1
0010	ALPHA	DISPARITY	NO	2
0011	DISPARITY	DISPARITY	NO	2
0100	ALPHA	DISPARITY	DISPARITY	3
0101	DEPTH	NO	NO	1
0110	ALPHA	DEPTH	NO	2
0111	TEXTURE	NO	NO	1
1000	USER DEFINED	NO	NO	1
1001	USER DEFINED	USER DEFINED	NO	2
1010	USER DEFINED	USER DEFINED	USER DEFINED	3
1011	ALPHA	USER DEFINED	NO	2
1100	ALPHA	USER DEFINED	USER DEFINED	3
1101-1111	t.b.d.	t.b.d.	t.b.d.	t.b.d.

vop_time_increment_resolution: This is a 16-bit unsigned integer that indicates the number of evenly spaced subintervals, called ticks, within one modulo time. One modulo time represents the fixed interval of one second. The value zero is forbidden.

fixed_vop_rate: This is a one-bit flag which indicates that all VOPs are coded with a fixed VOP rate. It shall only be '1' if and only if all the distances between the display time of any two successive VOPs in the display order in the video object layer are constant. In this case, the VOP rate can be derived from the fixed_VOP_time_increment. If it is '0' the display time between any two successive VOPs in the display order can be variable thus indicated by the time stamps provided in the VOP header.

fixed_vop_time_increment: This value represents the number of ticks between two successive VOPs in the display order. The length of a tick is given by VOP_time_increment_resolution. It can take a value in the range of [0,VOP_time_increment_resolution). The number of bits representing the value is calculated as the minimum number of unsigned integer bits required to represent the above range. fixed_VOP_time_increment shall only be present if fixed_VOP_rate is '1' and its value must be identical to the constant given by the distance between the display time of any two successive VOPs in the display order. In this case, the fixed VOP rate is given as (VOP_time_increment_resolution / fixed_VOP_time_increment). A zero value is forbidden.

EXAMPLE

VOP time = tick × vop_time_increment

= vop_time_increment / vop_time_increment_resolution

Table 6-15 -- Examples of vop_time_increment_resolution, fix_vop_time_increment, and vop_time_increment

Fixed VOP rate = 1/VOP time	vop_time_increment_ resolution	fixed_vop_time_ increment	vop_time_increment
15Hz	15	1	0, 1, 2, 3, 4,…
7.5Hz	15	2	0, 2, 4, 6, 8,…
29.97..Hz	30000	1001	0, 1001, 2002, 3003,…
59.94..Hz	60000	1001	0, 1001, 2002, 3003,…

video_object_layer_width: The video_object_layer_width is a 13-bit unsigned integer representing the width of the displayable part of the luminance component in pixel units. The width of the encoded luminance component of VOPs in macroblocks is (video_object_layer_width+15)/16. The displayable part is left-aligned in the encoded VOPs. A zero value is forbidden.

video_object_layer_height: The video_object_layer_height is a 13-bit unsigned integer representing the height of the displayable part of the luminance component in pixel units. The height of the encoded luminance component of VOPs in macroblocks is (video_object_layer_height+15)/16. The displayable part is top-aligned in the encoded VOPs. A zero value is forbidden.

interlaced: This is a 1 bit flag which, when set to "1" indicates that the VOP may contain interlaced video. When this flag is set to "0", the VOP is of non-interlaced (or progressive) format.

obmc_disable: This is a one-bit flag which when set to '1' disables overlapped block motion compensation.

sprite_enable: When video_object_layer_verid == '0001', this is a one-bit flag which when set to '1' indicates the usage of static (basic or low latency) sprite coding. When video_object_layer_verid == '0002', this is a two-bit unsigned integer which indicates the usage of static sprite coding or global motion compensation (GMC). Table V2 - 2 shows the meaning of various codewords. An S-VOP with sprite_enable == "GMC" is referred to as an S (GMC)-VOP in this document.

Table V2 - 2 -- Meaning of sprite_enable codewords

sprite_enable (video_object_layer_verid == '0001')	sprite_enable (video_object_layer_verid == '0002')	Sprite Coding Mode
0	00	sprite not used
1	01	static (Basic/Low Latency)
–	10	GMC (Global Motion Compensation)
–	11	Reserved

sprite_width: This is a 13-bit unsigned integer which identifies the horizontal dimension of the sprite.

sprite_height: This is a 13-bit unsigned integer which identifies the vertical dimension of the sprite.

sprite_left_coordinate: This is a 13-bit signed integer which defines the left edge of the sprite. The value of sprite_left_coordinate shall be divisible by two.

sprite_top_coordinate: This is a 13-bit signed integer which defines the top edge of the sprite. The value of sprite_top_coordinate shall be divisible by two.

no_of_sprite_warping_points: This is a 6-bit unsigned integer which represents the number of points used in sprite warping. When its value is 0 and when sprite_enable is set to 'static' or 'GMC', warping is identity (stationary sprite) and no coordinates need to be coded. When its value is 4, a perspective transform is used. When its value is 1,2 or 3, an affine transform is used. Further, the case of value 1 is separated as a special case from that of values 2 or 3. Table 6-16 shows the various choices. Note that the value of 4 is disallowed when sprite_enable == 'GMC'.

Table 6-16 -- Number of point and implied warping function

Number of points	warping function
0	Stationary
1	Translation
2,3	Affine
4	Perspective
5-63	Reserved

sprite_warping_accuracy – This is a 2-bit code which indicates the quantisation accuracy of motion vectors used in the warping process for sprites and GMC. Table 6-17 shows the meaning of various codewords

Table 6-17 -- Meaning of sprite warping accuracy codewords

code	sprite_warping_accuracy
00	½ pixel
01	¼ pixel
10	1/8 pixel
11	1/16 pixel

sprite_brightness_change: This is a one-bit flag which when set to '1' indicates a change in brightness during sprite warping, alternatively, a value of '0' means no change in brightness.

low_latency_sprite_enable: This is a one-bit flag which when set to "1" indicates the presence of low_latency sprite, alternatively, a value of "0" means basic sprite.

not_8_bit: This one bit flag is set when the video data precision is not 8 bits per pixel and visual object type is N-bit.

sadct_disable: This is a one-bit flag specifying the inverse transforms to be used for texture decoding. If 'sadct_disable' is set to '1', standard inverse DCT as described in version 1 is applied to all 8x8-blocks. When set to '0', flag 'sadct_disable' indicates that different types of inverse DCT are used in an adaptive way: standard inverse DCT is applied to those 8x8-blocks where all 64 pels are opaque, whereas inverse shape-adaptive DCT (SA-DCT) and inverse ΔDC-SA-DCT – an extended version of SA-DCT - are used in inter- and intra-coded 8x8-blocks with at least one transparent and one opaque pel.

quant_precision: This field specifies the number of bits used to represent quantiser parameters. Values between 3 and 9 are allowed. When not_8_bit is zero, and therefore quant_precision is not transmitted, it takes a default value of 5.

bits_per_pixel: This field specifies the video data precision in bits per pixel. It may take different values for different video object layers within a single video object. A value of 12 in this field would indicate 12 bits per pixel. This field may take values between 4 and 12. When not_8_bit is zero and bits_per_pixel is not present, the video data precision is always 8 bits per pixel, which is equivalent to specifying a value of 8 in this field. The same number of bits per pixel is used in the luminance and two chrominance planes. The alpha plane, used to specify shape of video objects, is always represented with 8 bits per pixel.

no_gray_quant_update: This is a one bit flag which is set to '1' when a fixed quantiser is used for the decoding of grayscale alpha data. When this flag is set to '0', the grayscale alpha quantiser is updated on every macroblock by generating it anew from the luminance quantiser value, but with an appropriate scale factor applied. See the description in subclause 7.5.4.3.

composition_method: This is a one bit flag which indicates which blending method is to be applied to the video object in the compositor. When set to '0', cross-fading shall be used. When set to '1', additive mixing shall be used. See subclause 7.5.4.6.

linear_composition: This is a one bit flag which indicates the type of signal used by the compositing process. When set to '0', the video signal in the format from which it was produced by the video decoder is used. When set to '1', linear signals are used. See subclause 7.5.4.6.

quant_type: This is a one-bit flag which when set to '1' that the first inverse quantisation method and when set to '0' indicates that the second inverse quantisation method is used for inverse quantisation of the DCT coefficients. Both inverse quantisation methods are described in subclause 7.4.4. For the first inverse quantisation method, two matrices are used, one for intra blocks the other for non-intra blocks.

The default matrix for intra blocks is:

8	17	18	19	21	23	25	27
17	18	19	21	23	25	27	28
20	21	22	23	24	26	28	30
21	22	23	24	26	28	30	32
22	23	24	26	28	30	32	35
23	24	26	28	30	32	35	38
25	26	28	30	32	35	38	41
27	28	30	32	35	38	41	45

The default matrix for non-intra blocks is:

16	17	18	19	20	21	22	23
17	18	19	20	21	22	23	24
18	19	20	21	22	23	24	25
19	20	21	22	23	24	26	27
20	21	22	23	25	26	27	28
21	22	23	24	26	27	28	30
22	23	24	26	27	28	30	31
23	24	25	27	28	30	31	33

load_intra_quant_mat This is a one-bit flag which is set to '1' when intra_quant_mat follows. If it is set to '0' then there is no change in the values that shall be used.

intra_quant_mat: This is a list of 2 to 64 eight-bit unsigned integers. The new values are in zigzag scan order and replace the previous values. A value of 0 indicates that no more values are transmitted and the remaining, non-transmitted values are set equal to the last non-zero value. The first value shall always be 8 and is not used in the decoding process.

load_nonintra_quant_mat This is a one-bit flag which is set to '1' when nonintra_quant_mat follows. If it is set to '0' then there is no change in the values that shall be used.

nonintra_quant_mat: This is a list of 2 to 64 eight-bit unsigned integers. The new values are in zigzag scan order and replace the previous values. A value of 0 indicates that no more values are transmitted and the remaining, non-transmitted values are set equal to the last non-zero value. The first value shall not be 0.

load_intra_quant_mat_grayscale: This is a one-bit flag which is set to '1' when intra_quant_mat_grayscale follows. If it is set to '0' then there is no change in the quantisation matrix values that shall be used.

intra_quant_mat_grayscale: This is a list of 2 to 64 eight-bit unsigned integers defining the grayscale intra alpha quantisation matrix to be used. The semantics and the default quantisation matrix are identical to those of intra_quant_mat.

load_nonintra_quant_mat_grayscale: This is a one-bit flag which is set to '1' when nonintra_quant_mat_grayscale[i] follows for grayscale alpha or auxiliary component i=0,1,2. If it is set to '0' then there is no change in the quantisation matrix values that shall be used.

intra_quant_mat_grayscale[i]: This is a list of 2 to 64 eight-bit unsigned integers defining the grayscale intra alpha quantisation matrix to be used for grayscale alpha or auxiliary component i=0,1,2. The semantics and the default quantisation matrix are identical to those of intra_quant_mat.

nonintra_quant_mat_grayscale: This is a list of 2 to 64 eight-bit unsigned integers defining the grayscale nonintra alpha quantisation matrix[i] to be used for grayscale alpha or auxiliary component i=0,1,2. The semantics and the default quantisation matrix are identical to those of nonintra_quant_mat.

nonintra_quant_mat_grayscale[i]: This is a list of 2 to 64 eight-bit unsigned integers defining the grayscale nonintra alpha quantisation matrix to be used for grayscale alpha or auxiliary component i=0,1,2. The semantics and the default quantisation matrix are identical to those of nonintra_quant_mat.

quarter_sample: This is a one-bit flag which when set to '0' indicates that half sample mode and when set to '1' indicates that quarter sample mode shall be used for motion compensation of the luminance component.

complexity_estimation_disable: This is a one-bit flag which, when set to '1', disables complexity estimation header in each VOP.

estimation_method: Setting of the of the estimation method, it is „00" for Version 1 and "01" for version 2.

shape_complexity_estimation_disable: This is a one-bit flag which when set to '1' disables shape complexity estimation.

opaque: Flag enabling transmission of the number of luminance and chrominance blocks coded using opaque coding mode in % of the total number of blocks (bounding rectangle).

transparent: Flag enabling transmission of the number of luminance and chrominance blocks coded using transparent mode in % of the total number of blocks (bounding rectangle).

intra_cae: Flag enabling transmission of the number of luminance and chrominance blocks coded using IntraCAE coding mode in % of the total number of blocks (bounding rectangle).

inter_cae: Flag enabling transmission of the number of luminance and chrominance blocks coded using InterCAE coding mode in % of the total number of blocks (bounding rectangle).

no_update: Flag enabling transmission of the number of luminance and chrominance blocks coded using no update coding mode in % of the total number of blocks (bounding rectangle).

upsampling: Flag enabling transmission of the number of luminance and chrominance blocks which need upsampling from 4-4- to 8-8 block dimensions in % of the total number of blocks (bounding rectangle).

version2_complexity_estimation_disable: Flag to disable version 2 parameter set.

sadct: Flag enabling transmission of the number of luminance and chrominance blocks coded using SADCT coding mode in % of the total number of blocks (bounding box). When estimation_method == '00' the value of sadct is set to '0'.

quarterpel: Flag enabling transmission of the number of luminance and chrominance block predicted by a quarter-pel vector on one or two dimensions (horizontal and vertical) in % of the total number of blocks (bounding box). When estimation_method == '00' the value of quarterpel is set to '0'.

texture_complexity_estimation_set_1_disable: Flag to disable texture parameter set 1.

intra_blocks: Flag enabling transmission of the number of luminance and chrominance Intra or Intra+Q coded blocks in % of the total number of blocks (bounding rectangle).

inter_blocks: Flag enabling transmission of the number of luminance and chrominance Inter and Inter+Q coded blocks in % of the total number of blocks (bounding rectangle).

inter4v_blocks: Flag enabling transmission of the number of luminance and chrominance Inter4V coded blocks in % of the total number of blocks (bounding rectangle).

not_coded_blocks: Flag enabling transmission of the number of luminance and chrominance Non Coded blocks in % of the total number of blocks (bounding rectangle).

texture_complexity_estimation_set_2_disable: Flag to disable texture parameter set 2.

dct_coefs: Flag enabling transmission of the number of DCT coefficients % of the maximum number of coefficients (coded blocks).

dct_lines: Flag enabling transmission of the number of DCT8x1 in % of the maximum number of DCT8x1 (coded blocks).

vlc_symbols: Flag enabling transmission of the average number of VLC symbols for macroblock.

vlc_bits: Flag enabling transmission of the average number of bits for each symbol.

motion_compensation_complexity_disable: Flag to disable motion compensation parameter set.

apm (Advanced Prediction Mode): Flag enabling transmission of the number of luminance block predicted using APM in % of the total number of blocks for VOP (bounding rectangle).

npm (Normal Prediction Mode): Flag enabling transmission of the number of luminance and chrominance blocks predicted using NPM in % of the total number of luminance and chrominance for VOP (bounding rectangle).

interpolate_mc_q: Flag enabling transmission of the number of luminance and chrominance interpolated blocks in % of the total number of blocks for VOP (bounding rectangle).

forw_back_mc_q: Flag enabling transmission of the number of luminance and chrominance predicted blocks in % of the total number of blocks for VOP (bounding rectangle).

halfpel2: Flag enabling transmission of the number of luminance and chrominance block predicted by a half-pel vector on one dimension (horizontal or vertical) in % of the total number of blocks (bounding rectangle).

halfpel4: Flag enabling transmission of the number of luminance and chrominance block predicted by a half-pel vector on two dimensions (horizontal and vertical) in % of the total number of blocks (bounding rectangle).

resync_marker_disable: This is a one-bit flag which when set to '1' indicates that there is no resync_marker in coded VOPs. This flag can be used only for the optimization of the decoder operation. Successful decoding can be carried out without taking into account the value of this flag.

data_partitioned: This is a one-bit flag which when set to '1' indicates that the macroblock data is rearranged differently, specifically, motion vector data is separated from the texture data (i.e., DCT coefficients).

reversible_vlc: This is a one-bit flag which when set to '1' indicates that the reversible variable length tables (Table B-23, Table B-24 and Table B-25) should be used when decoding DCT coefficients. These tables can only be used when data_partition flag is enabled. Note that this flag shall be treated as '0' in B-VOPs. Use of escape sequence (Table B-24 and Table B-25) for encoding the combinations listed in Table B-23 is prohibited.

newpred_enable: This is a one-bit flag which, when set to '1', indicates that the NEWPRED mode is enabled. When video_object_layer_verid is equal to '0001', and therefore newpred_enable is not transmitted, it takes a default value of zero.

requested_upstream_message_type: This is a two-bits flag which indicates which type of upstream message is needed by the encoder. The syntax and semantics of the upstream message are described in subclause 6.2.12 and 6.3.12.

01: need NP_ACK message to be returned for each NEWPRED segment
10: need NP_NACK message to be returned for each NEWPRED segment
11: need both NP_ACK and NP_NACK messages to be returned for each NEWPRED segment
00: reserved

newpred_segment_type: This is a one-bits flag which indicates the unit of selecting reference VOP (NEWPRED segment).

0: Video Packet
1: VOP

reduced_resolution_vop_enable: This is a one-bit flag which indicates that the reduced resolution vop tool is enabled when set to '1'. When video_object_layer_verid is equal to '0001', and therefore reduced_resolution_vop_enable is not transmitted, it takes a default value of zero.

scalability: This is a one-bit flag which when set to '1' indicates that the current layer uses scalable coding. If the current layer is used as base-layer then this flag is set to '0'. Additionally, this flag shall be set to '0' for S(GMC)-VOPs.

hierarchy_type: The hierarcical relation between the associated hierarchy layer and its hierarchy embedded layer is defined as shown in Table 6-18.

Table 6-18 -- Code table for hierarchy_type

Description	Code
ISO/IEC 14496-2 Spatial Scalability	0
ISO/IEC 14496-2 Temporal Scalability	1

ref_layer_id: This is a 4-bit unsigned integer with value between 0 and 15. It indicates the layer to be used as reference for prediction(s) in the case of scalability.

ref_layer_sampling_direc: This is a one-bit flag which when set to '1' indicates that the resolution of the reference layer (specified by reference_layer_id) is higher than the resolution of the layer being coded. If it is set to '0' then the reference layer has the same or lower resolution than the resolution of the layer being coded.

hor_sampling_factor_n: This is a 5-bit unsigned integer which forms the numerator of the ratio used in horizontal spatial resampling in scalability. The value of zero is forbidden.

hor_sampling_factor_m: This is a 5-bit unsigned integer which forms the denominator of the ratio used in horizontal spatial resampling in scalability. The value of zero is forbidden.

vert_sampling_factor_n: This is a 5-bit unsigned integer which forms the numerator of the ratio used in vertical spatial resampling in scalability. The value of zero is forbidden.

vert_sampling_factor_m: This is a 5-bit unsigned integer which forms the denominator of the ratio used in vertical spatial resampling in scalability. The value of zero is forbidden.

enhancement_type: This is a 1-bit flag which is set to '1' when the current layer enhances the partial region of the reference layer. If it is set to '0' then the current layer enhances the entire region of the reference layer. The default value of this flag is '0'.

use_ref_shape: This is one bit flag which indicate procedure to decode binary shape for spatial scalability. If it is set to '0', scalable shape coding should be used. If it is set to '1' and enhancement_type is set to '0', no shape data is decoded and up-sampled binary shape of base layer should be used for enhancement layer. If enhancement_type is set to '1' and this flag is set to '1',binary shape of enhancement layer should be decoded as

the same non-scalable decoding process. When video_object_layer_verid == '0001', the value of use_ref_shape_ is set to '1'.

use_ref_texture: When this one bit is set, no update for texture is done. Instead, the available texture in the layer denoted by ref_layer_id will be used.

shape_hor_sampling_factor_n: This is a 5-bit unsigned integer which forms the numerator of the ratio used in horizontal spatial resampling in shape scalability. The value of zero is forbidden.

shape_hor_sampling_factor_m: This is a 5-bit unsigned integer which forms the denominator of the ratio used in horizontal spatial resampling in shape scalability. The value of zero is forbidden.

shape_vert_sampling_factor_n: This is a 5-bit unsigned integer which forms the denominator of the ratio used in vertical spatial resampling in shape scalability. The value of zero is forbidden.

shape_vert_sampling_factor_m: This is a 5-bit unsigned integer which forms the denominator of the ratio used in vertical spatial resampling in shape scalability. The value of zero is forbidden.

6.3.4 Group of Video Object Plane

group_of_vop_start_code: The group_of_vop_start_code is the bit string '000001B3' in hexadecimal. It identifies the beginning of a GOV header.

time_code: This is a 18-bit integer containing the following: time_code_hours, time_code_minutes, marker_bit and time_code_seconds as shown in Table 6-19. The parameters correspond to those defined in the IEC standard publication 461 for "time and control codes for video tape recorders". The time code specifies the modulo part (i.e. the full second units) of the time base for the first object plane (in display order) after the GOV header.

Table 6-19 -- Meaning of time_code

time_code	range of value	No. of bits	Mnemonic
time_code_hours	0 - 23	5	uimsbf
time_code_minutes	0 - 59	6	uimsbf
marker_bit	1	1	bslbf
time_code_seconds	0 - 59	6	uimsbf

closed_gov: This is a one-bit flag which indicates the nature of the predictions used in the first consecutive B-VOPs (if any) immediately following the first coded I-VOP after the GOV header .The closed_gov is set to '1' to indicate that these B-VOPs have been encoded using only backward prediction or intra coding. This bit is provided for use during any editing which occurs after encoding. If the previous pictures have been removed by editing, broken_link may be set to '1' so that a decoder may avoid displaying these B-VOPs following the first I-VOP following the group of plane header. However if the closed_gov bit is set to '1', then the editor may choose not to set the broken_link bit as these B-VOPs can be correctly decoded.

broken_link: This is a one-bit flag which shall be set to '0' during encoding. It is set to '1' to indicate that the first consecutive B-VOPs (if any) immediately following the first coded I-VOP following the group of plane header may not be correctly decoded because the reference frame which is used for prediction is not available (because of the action of editing). A decoder may use this flag to avoid displaying frames that cannot be correctly decoded.

6.3.5 Video Object Plane and Video Plane with Short Header

vop_start_code: This is the bit string ' 000001B6' in hexadecimal. It marks the start of a video object plane.

vop_coding_type: The vop_coding_type identifies whether a VOP is an intra-coded VOP (I), predictive-coded VOP (P), bidirectionally predictive-coded VOP (B) or sprite coded VOP (S). The meaning of vop_coding_type is defined in Table 6-20.

Table 6-20 -- Meaning of vop_coding_type

vop_coding_type	coding method
00	intra-coded (I)
01	predictive-coded (P)
10	bidirectionally-predictive-coded (B)
11	sprite (S)

modulo_time_base: This value represents the local time base in one second resolution units (1000 milliseconds). It consists of a number of consecutive '1' followed by a '0'. Each '1' represents a duration of one second that have elapsed. For I-, S(GMC)-, and P-VOPs of a non scalable bitstream and the base layer of a scalable bitstream, the number of '1' s indicate the number of seconds elapsed since the synchronization point marked by time_code of the previous GOV header or by modulo_time_base of the previously decoded I-, S(GMC)-, or P-VOP, in decoding order. For B-VOP of non scalable bitstream and base layer of scalable bitstream, the number of '1' sindicate the number of seconds elapsed since the synchronization point marked in the previous GOV header, I-VOP, S(GMC)-VOP, or P-VOP, in display order. For I-, P-, or B-VOPs of enhancement layer of scalable bitstream, the number of '1' s indicate the number of seconds elapsed since the synchronization point marked in the previous GOV header, I-VOP, P-VOP, or B-VOP, in display order.

vop_time_increment: This value represents the absolute vop_time_increment from the synchronization point marked by the modulo_time_base measured in the number of clock ticks. It can take a value in the range of [0,vop_time_increment_resolution). The number of bits representing the value is calculated as the minimum number of unsigned integer bits required to represent the above range. The local time base in the units of seconds is recovered by dividing this value by the vop_time_increment_resolution.

vop_coded: This is a 1-bit flag which when set to '0' indicates that no subsequent data exists for the VOP. In this case, the following decoding rules apply: If binary shape or alpha plane does exist for the VOP (i.e. video_object_layer_shape != "rectangular"), it shall be completely transparent. If binary shape or alpha plane does not exist for the VOP (i.e. video_object_layer_shape == "rectangular"), the luminance and chrominance planes of the reconstructed VOP shall be filled with the forward reference VOP as defined in subclause 7.6.7.

vop_rounding_type: This is a one-bit flag which signals the value of the parameter rounding_control used for pixel value interpolation in motion compensation for P- and S(GMC)- VOPs. When this flag is set to '0', the value of rounding_control is 0, and when this flag is set to '1', the value of rounding_control is 1. When vop_rounding_type is not present in the VOP header, the value of rounding_control is 0.

vop_reduced_resolution: This single bit flag signals whether the VOP is encoded in spatialy reduced resolution or not. When vop_reduced_resolution is not transmitted, it takes a default value of zero. When this flag is set to '1', the VOP is encoded spatialy reduced resolution and referred as Reduced Resolution VOP. Reduced Resolution VOP shall be decoded by the texture decoding process including upsampling of IDCT output, and motion compensation decoding process of Reduced Resolution VOP. In this case, the width of the encoded luminance component of the vop in macroblocks is (video_object_layer_width+31)/32 and the the height of the encoded luminance component of the vop in macroblocks is (video_object_layer_height+31)/32. The displayable part is top and left-aligned in the encoded vop. This flag shall not be the '1' when the (video_object_layer_width+15)/16 is not the multiple of 2 or the (video_object_layer_width+15)/16 is not the multiple of 2. When this flag is set to "0" or this flag is not present, the VOP is encoded in normal spatial resolution and shall be decoded by the normal decoding process.

vop_width: This is a 13-bit unsigned integer which specifies the horizontal size, in pixel units, of the rectangle that includes the VOP. The width of the encoded luminance component of VOP in macroblocks is (vop_width+15)/16. The rectangle part is left-aligned in the encoded VOP. A zero value is forbidden.

vop_height: This is a 13-bit unsigned integer which specifies the vertical size, in pixel units, of the rectangle that includes the VOP. The height of the encoded luminance component of VOP in macroblocks is (vop_height+15)/16. The rectangle part is top-aligned in the encoded VOP. A zero value is forbidden.

vop_horizontal_mc_spatial_ref: This is a 13-bit signed integer which specifies, in pixel units, the horizontal position of the top left of the rectangle defined by horizontal size of vop_width. The value of vop_horizontal_mc_spatial_ref shall be divisible by two. This is used for decoding and for picture composition.

vop_vertical_mc_spatial_ref: This is a 13-bit signed integer which specifies, in pixel units, the vertical position of the top left of the rectangle defined by vertical size of vop_width. The value of vop_vertical_mc_spatial_ref shall be divisible by two for progressive and divisible by four for interlaced motion compensation. This is used for decoding and for picture composition.

background_composition: This flag only occurs when scalability flag has a value of "1. This flag is used in conjunction with enhancement_type flag. If enhancement_type is "1", hierarchy_type is "1", and this flag is "1", background composition specified in subclause 8.1 is performed. If enhancement type is "1", hierarchy_type is "1" and this flag is "0", any method can be used to make a background for the enhancement layer.

When hierarchy_type is "0", video_object_layer_shape is "binary" (object based spatial scalability), enhancement_type is " 1", and background_compositin flag is "1", background composition specified in subclause 8.4 is performed and when enhancement_type is "1" and this flag is "0", any method can be used to make a background for the enhancement layer.

change_conv_ratio_disable: This is a 1-bit flag which when set to '1' indicates that conv_ratio is not sent at the macroblock layer and is assumed to be 1 for all the macroblocks of the VOP. When set to '0', the conv_ratio is coded at macroblock layer.

vop_constant_alpha: This bit is used to indicate the presence of vop_constant_alpha_value. When this is set to one, vop_constant_alpha_value is included in the bitstream.

vop_constant_alpha_value: This is an unsigned integer which indicates the scale factor to be applied as a post processing phase of binary or grayscale shape decoding. See subclause 7.5.4.2.

intra_dc_vlc_thr: This is a 3-bit code allows a mechanism to switch between two VLC's for coding of Intra DC coefficients as per Table 6-21.

Table 6-21 -- Meaning of intra_dc_vlc_thr

index	meaning of intra_dc_vlc_thr	code
0	Use Intra DC VLC for entire VOP	000
1	Switch to Intra AC VLC at running Qp >=13	001
2	Switch to Intra AC VLC at running Qp >=15	010
3	Switch to Intra AC VLC at running Qp >=17	011
4	Switch to Intra AC VLC at running Qp >=19	100
5	Switch to Intra AC VLC at running Qp >=21	101
6	Switch to Intra AC VLC at running Qp >=23	110
7	Use Intra AC VLC for entire VOP	111

Where running Qp is defined as the DCT quantisation parameter for luminance and chrominance used for immediately previous coded macroblock, except for the first coded macroblock in a VOP or a video packet. At the first coded macroblock in a VOP or a video packet, the running Qp is defined as the quantisation parameter value for the current macroblock.

top_field_first: This is a 1-bit flag which when set to "1" indicates that the top field (i.e., the field containing the top line) of reconstructed VOP is the first field to be displayed (output by the decoding process). When top_field_first is set to "0" it indicates that the bottom field of the reconstructed VOP is the first field to be displayed.

alternate_vertical_scan_flag: This is a 1-bit flag which when set to "1" indicates the use of alternate vertical scan for interlaced VOPs.

sprite_transmit_mode: This is a 2-bit code which signals the transmission mode of the sprite object. At video object layer initialization, the code is set to "piece" mode. When all object and quality update pieces are sent for the entire video object layer, the code is set to the " stop" mode. When an object piece is sent, the code is set to "piece" mode. When an update piece is being sent, the code is set to the "update" mode. When all sprite object pieces andquality update pieces for the current VOP are sent, the code is set to "pause" mode. Table 6-22 shows the different sprite transmit modes.

Table 6-22 -- Meaning of sprite transmit modes

code	sprite_transmit_mode
00	stop
01	piece
10	update
11	pause

vop_quant: This is an unsigned integer which specifies the absolute value of quant to be used for dequantizing the macroblock until updated by any subsequent dquant, dbquant, or quant_scale. The length of this field is specified by the value of the parameter quant_precision. The default length is 5bits which carries the binary representation of quantizer values from 1 to 31 in steps of 1.

vop_alpha_quant[i]: This is a an unsigned integer which specifies the absolute value of the initial alpha plane quantiser to be used for dequantising macroblock grayscale alpha data in alpha or auxiliary component i=0,1,2. The alpha plane quantiser cannot be less than 1.

vop_fcode_forward: This is a 3-bit unsigned integer taking values from 1 to 7; the value of zero is forbidden. It is used in decoding of motion vectors.

vop_fcode_backward: This is a 3-bit unsigned integer taking values from 1 to 7; the value of zero is forbidden. It is used in decoding of motion vectors.

vop_shape_coding_type: This is a 1 bit flag which specifies whether inter shape decoding is to be carried out for the current P, B, or S(GMC)-VOP. If vop_shape_coding_type is equal to '0', intra shape decoding is carried out, otherwise inter shape decoding is carried out.

Coded data for the top-left macroblock of the bounding rectangle of a VOP shall immediately follow the VOP header, followed by the remaining macroblocks in the bounding rectangle in the conventional left-to-right, top-to-bottom scan order. Video packets shall also be transmitted following the conventional left-to-right, top-to-bottom macroblock scan order. The last MB of one video packet is guaranteed to immediately precede the first MB of the following video packet in the MB scan order.

load_backward_shape: This is a one-bit flag which when set to '1' implies that the backward shape of the previous VOP in the same layer is copied to the forward shape for the current VOP and the backward shape of the current VOP is decoded from the bitstream. When this flag is set to '0', the forward shape of the previous VOP is copied to the forward_shape of the current VOP and the backward shape of the previous VOP in the same layer is copied to the backward shape of the current VOP. This flag shall be '1' when (1) background_composition is '1' and vop_coded of the previous VOP in the same layer is '0' or (2) background_composition is '1' and the current VOP is the first VOP in the current layer.

If hierarchy_type is "0" and video_object_layer_shape is " binary" (object based spatial scalability), then this flag shall be set to "0".

backward_shape_width: This is a 13-bit unsigned integer which specifies the horizontal size, in pixel units, of the rectangle that includes the backward shape. A zero value is forbidden.

backward_shape_height: This is a 13-bit unsigned integer which specifies the vertical size, in pixel units, of the rectangle that includes the backward shape. A zero value is forbidden.

backward_shape_horizontal_mc_spatial_ref: This is a 13-bit signed integer which specifies, in pixel units, the horizontal position of the top left of the rectangle that includes the backward shape. This is used for decoding and for picture composition.

backward_shape_vertical_mc_spatial_ref: This is a 13-bit signed integer which specifies, in pixel units, the vertical position of the top left of the rectangle that includes the backward shape. This is used for decoding and for picture composition.

backward_shape(): The decoding process of the backward shape is identical to the decoding process for the shape of I-VOP with binary only mode (video_object_layer_shape = "10").

load_forward_shape: This is a one-bit flag which when set to '1' implies that the forward shape is decoded from the bitstream. This flag shall be '1' when (1) background_composition is '1' and vop_coded of the previous VOP in the same layer is '0' or (2) background_composition is '1' and the current VOP is the first VOP in the current layer.

forward_shape_width: This is a 13-bit unsigned integer which specifies the horizontal size, in pixel units, of the rectangle that includes the forward shape. A zero value is forbidden.

forward_shape_height: This is a 13-bit unsigned integer which specifies the vertical size, in pixel units, of the rectangle that includes the forward shape. A zero value is forbidden.

forward_shape_horizontal_mc_spatial_ref: This is a 13-bit signed integer which specifies, in pixel units, the horizontal position of the top left of the rectangle that includes the forward shape. This is used for decoding and for picture composition.

forward_shape_vertical_mc_spatial_ref: This is a 13-bit signed integer which specifies, in pixel units, the vertical position of the top left of the rectangle that includes the forward shape. This is used for decoding and for picture composition.

forward_shape(): The decoding process of the backward shape is identical to the decoding process for the shape of I-VOP with binary only mode (video_object_layer_shape = "10").

ref_select_code: This is a 2-bit unsigned integer which specifies prediction reference choices for P- and B-VOPs in enhancement layer with respect to decoded reference layer identified by ref_layer_id. The meaning of allowed values is specified in Table 7-13 and Table 7-14.

resync_marker: This is a binary string of at least 16 zero's followed by a one'0 0000 0000 0000 0001'. For an I-VOP or a VOP where video_object_layer_shape has the value "binary_only", the resync marker is 16 zeros followed by a one. The length of this resync marker is dependent on the value of vop_fcode_forward, for a P-VOP or a S(GMC)-VOP, and the larger value of either vop_fcode_forward and vop_fcode_backward for a B-VOP. The relationship between the length of the resync_marker and appropriate fcode is given by 16 + fcode. The resync_marker is (15+fcode) zeros followed by a one. It is only present when resync_marker_disable flag is set to '0'. A resync marker shall only be located immediately before a macroblock and aligned with a byte

macroblock_number: This is a variable length code with length between 1 and 14 bits. It identifies the macroblock number within a VOP. The number of the top-left macroblock in a VOP shall be zero. The macroblock number increases from left to right and from top to bottom. The actual length of the code depends on the total number of macroblocks in the VOP calculated according to Table 6-23, the code itself is simply a binary representation of the macroblock number. If reduced_resolution_vop_enable is equal to one, the length of macroblock_number code shall be determined by as if the total number of macroblocks in a VOP in Table 6-23 be equal to ((video_object_layer_width+15)/16) * ((video_object_layer_height+15)/16).

Table 6-23 -- Length of macroblock_number code

length of macroblock_number code	((vop_width+15)/16) * ((vop_height+15)/16)
1	1-2
2	3-4
3	5-8
4	9-16
5	17-32
6	33-64
7	65-128
8	129-256
9	257-512
10	513-1024
11	1025-2048
12	2049-4096
13	4097-8192
14	8193-16384

quant_scale: This is an unsigned integer which specifies the absolute value of quant to be used for dequantizing the macroblock of the video packet until updated by any subsequent dquant. The length of this field is specified by the value of the parameter quant_precision. The default length is 5-bits.

header_extension_code: This is a 1-bit flag which when set to '1' indicates the prescence of additional fields in the header. When header_extension_code is is se to '1', modulo_time_base, vop_time_increment and vop_coding_type are also included in the video packet header. Furthermore, if the vop_coding_type is equal to either a P, S or B VOP, the appropriate fcodes are also present. Additionally, if the current VOP is an S(GMC)-VOP, sprite_trajectory() is included. And if reduced_resolution_vop_enable is equal to one, vop_reduced_resolution is also present.

use_intra_dc_vlc: The value of this internal flag is set to 1 when the values of intra_dc_thr and the DCT quantiser for luminance and chrominace indicate the usage of the intra DC VLCs shown in Table B-13 - Table B-15 for the decoding of intra DC coefficients. Otherwise, the value of this flag is set to 0.

motion_marker: This is a 17-bit binary string '1 1111 0000 0000 0001'. It is only present when the data_partitioned flag is set to '1'.In the data partitioning mode, a motion_marker is inserted after the motion data (prior to the texture data). The motion_marker is unique from the motion data and enables the decoder to determine when all the motion information has been received correctly.

dc_marker: This is a 19 bit binary string '110 1011 0000 0000 0001'. It is present when the data_partitioned flag is set to '1'. It is used for I-VOPs. In the data partitioning mode, a dc_marker is inserted into the bitstream after the mcbpc, dquant and dc data but before the ac_pred flag and remaining texture information.

vop_id: This indicates the ID of VOP which is incremented by 1 whenever a VOP is encoded. All '0' value is skipped in vop_id in order to prevent the resync_marker emulation. The bit length of vop_id is the smaller value between (the length of vop_time_increment) + 3 and 15.

vop_id_for_prediction_indication: This is a one-bit flag which indicates the existence of the following vop_id_for_prediction field.

0: vop_id_for_prediction does not exist.
1: vop_id_for_prediction does exist.

vop_id_for_prediction: This indicates the vop_id of the VOP which is used as the reference VOP of the decoding of the current VOP or Video Packet. When this field exists, the reference VOP shall be replaced with the indicated one. This VOP which is used for prediction may be changed according to the backward channel message. If this field does not exist, the previous VOP is used for prediction, or all MBs in the VOP or Video Packet are coded to Intra MB.

6.3.5.1 Definition of DCECS variable values

The semantic of all complexity estimation parameters is defined at the VO syntax level. DCECS variables represent % values. The actual % values have been converted to 8 bit words by normalization to 256. To each 8 bit word a binary 1 is added to prevent start code emulation (i.e 0% = '00000001' , 99.5% = '11111111' and is conventionally considered equal to one). The binary '00000000' string is a forbidden value. The only parameter expressed in their absolute value is the dcecs_vlc_bits parameter expressed as a 4 bit word.

dcecs_opaque: 8 bit number representing the % of luminance and chrominance blocks using opaque coding mode on the total number of blocks (bounding rectangle).

dcecs_transparent: 8 bit number representing the % of luminance and chrominance blocks using transparent coding mode on the total number of blocks (bounding rectangle).

dcecs_intra_cae: 8 bit number representing the % of luminance and chrominance blocks using IntraCAE coding mode on the total number of blocks (bounding rectangle).

dcecs_inter_cae: 8 bit number representing the % of luminance and chrominance blocks using InterCAE coding mode on the total number of blocks (bounding rectangle).

dcecs_no_update: 8 bit number representing the % of luminance and chrominance blocks using no update coding mode on the total number of blocks (bounding rectangle).

dcecs_upsampling: 8 bit number representing the % of luminance and chrominance blocks which need upsampling from 4-4- to 8-8 block dimensions on the total number of blocks (bounding rectangle).

dcecs_intra_blocks: 8 bit number representing the % of luminance and chrominance Intra or Intra+Q coded blocks on the total number of blocks (bounding rectangle).

dcecs_not_coded_blocks: 8 bit number representing the % of luminance and chrominance Non Coded blocks on the total number of blocks (bounding rectangle).

dcecs_dct_coefs: 8 bit number representing the % of the number of DCT coefficients on the maximum number of coefficients (coded blocks).

dcecs_dct_lines: 8 bit number representing the % of the number of DCT8x1 on the maximum number of DCT8x1 (coded blocks).

dcecs_vlc_symbols: 8 bit number representing the average number of VLC symbols for macroblock.

dcecs_vlc_bits: 4 bit number representing the average number of bits for each symbol.

dcecs_inter_blocks: 8 bit number representing the % of luminance and chrominance Inter and Inter+Q coded blocks on the total number of blocks (bounding rectangle).

dcecs_inter4v_blocks: 8 bit number representing the % of luminance and chrominance Inter4V coded blocks on the total number of blocks (bounding rectangle).

dcecs_apm (Advanced Prediction Mode): 8 bit number representing the % of the number of luminance block predicted using APM on the total number of blocks for VOP (bounding rectangle).

dcecs_npm (Normal Prediction Mode): 8 bit number representing the % of luminance and chrominance blocks predicted using NPM on the total number of luminance and chrominance blocks for VOP (bounding rectangle).

dcecs_forw_back_mc_q: 8 bit number representing the % of luminance and chrominance predicted blocks on the total number of blocks for VOP (bounding rectangle).

dcecs_halfpel2: 8 bit number representing the % of luminance and chrominance blocks predicted by a half-pel vector on one dimension (horizontal or vertical) on the total number of blocks (bounding rectangle).

dcecs_halfpel4: 8 bit number representing the % of luminance and chrominance blocks predicted by a half-pel vector on two dimensions (horizontal and vertical) on the total number of blocks (bounding rectangle).

dcecs_interpolate_mc_q 8 bit number representing the % of luminance and chrominance interpolated blocks in % of the total number of blocks for VOP (bounding rectangle).

dcecs_sadct 8 bit number representing the % of luminance and chrominance blocks coded using SADCT mode in % of the total number of blocks for VOP (bounding rectangle).

dcecs_quarterpel : 8 bit number representing the % of luminance and chrominance blocks predicted using quarter-pel vectors in % of the total number of blocks for VOP (bounding rectangle).

6.3.5.2 Video Plane with Short Header

video_plane_with_short_header() – This data structure contains a video plane using an abbreviated header format. Certain values of parameters shall have pre-defined and fixed values for any video_plane_with_short_header, due to the limited capability of signaling information in the short header format. These parameters having fixed values are shown in Table 6-24.

Table 6-24 -- Fixed Settings for video_plane_with_short_header()

Parameter	Value
video_object_layer_shape	“rectangular”
obmc_disable	1
quant_type	0
resync_marker_disable	1
data_partitioned	0
block_count	6
reversible_vlc	0
vop_rounding_type	0
vop_fcode_forward	1
vop_coded	1
interlaced	0
complexity_estimation_disable	1
use_intra_dc_vlc	0
scalability	0
not_8_bit	0
bits_per_pixel	8
colour_primaries	1
transfer_characteristics	1
matrix_coefficients	6

short_video_start_marker: This is a 22-bit start marker containing the value '0000 0000 0000 0000 1000 00'. It is used to mark the location of a video plane having the short header format. short_video_start_marker shall be byte aligned by the insertion of zero to seven zero-valued bits as necessary to achieve byte alignment prior to short_video_start_marker.

temporal_reference: This is an 8-bit number which can have 256 possible values. It is formed by incrementing its value in the previously transmitted video_plane_with_short_header() by one plus the number of non-transmitted pictures (at 30000/1001 Hz) since the previouslytransmitted picture. The arithmetic is performed with only the eight LSBs.

split_screen_indicator: This is a boolean signal that indicates that the upper and lower half of the decoded picture could be displayed side by side. This bit has no direct effect on the encoding or decoding of the video plane.

document_camera_indicator: This is a boolean signal that indicates that the video content of the vop is sourced as a representation from a document camera or graphic representation, as opposed to a view of natural video content. This bit has no direct effect on the encoding or decoding of the video plane.

full_picture_freeze_release: This is a boolean signal that indicates that resumption of display updates should be activated if the display of the video content has been frozen due to errors, packet losses, or for some other reason such as the receipt of a external signal. This bit has no direct effect on the encoding or decoding of the video plane.

source_format This is an indication of the width and height of the rectangular video plane represented by the video_plane_with_short_header. The meaning of this field is shown in Table 6-25. Each of these source formats has the same vop time increment resolution which is equal to 30000/1001 (approximately 29.97) Hz and the same width:height pixel aspect ratio (288/3):(352/4), which equals 12:11 in relatively prime numbers and which defines a CIF picture as having a width:height picture aspect ratio of 4:3.

Table 6-25 -- Parameters Defined by source_format Field

source_format value	Source Format Meaning	vop_width	vop_height	num_macroblocks_in_gob	num_gobs_in_vop
000	reserved	reserved	reserved	reserved	reserved
001	sub-QCIF	128	96	8	6
010	QCIF	176	144	11	9
011	CIF	352	288	22	18
100	4CIF	704	576	88	18
101	16CIF	1408	1152	352	18
110	reserved	reserved	reserved	reserved	reserved
111	reserved	reserved	reserved	reserved	reserved

picture_coding_type: This bit indicates the vop_coding_type. When equal to zero, the vop_coding_type is " I", and when equal to one, the vop_coding_type is "P".

four_reserved_zero_bits: This is a four-bit field containing bits which are reserved for future use and equal to zero.

pei: This is a single bit which, when equal to one, indicates the presence of a byte of psupp data following the pei bit.

psupp: This is an eight bit field which is present when pei is equal to one. The pei + psupp mechanism provides for a reserved method of later allowing the definition of backward-compatible data to be added to the bitstream. Decoders shall accept and discard psupp when pei is equal to one, with no effect on the decoding of the video data. The pei and psupp combination pair may be repeated if present. The ability for an encoder to add pei and psupp to the bitstream is reserved for future use.

gob_number: This is a five-bit number which indicates the location of video data within the video plane. A group of blocks (or GOB) contains a number of macroblocks in raster scanning order within the picture. For a given gob_number, the GOB contains the num_macroblocks_per_gob macroblocks starting with macroblock_number = gob_number * num_macroblocks_per_gob. The gob_number can either be read from the bitstream or inferred from the progress of macroblock decoding as shown in the syntax description pseudo-code.

num_gobs_in_vop: This is the number of GOBs in the vop. This parameter is derived from the source_format as shown in Table 6-25.

gob_layer(): This is a layer containing a fixed number of macroblocks in the vop. Which macroblocks which belong to each gob can be determined by gob_number and num_macroblocks_in_gob.

gob_resync_marker: This is a fixed length code of 17 bits having the value '0000 0000 0000 0000 1' which may optionally be inserted at the beginning of each gob_layer(). Its purpose is to serve as a type of resynchronization marker for error recovery in the bitstream. The gob_resync_marker codes may (and should) be byte aligned by inserting zero to seven zero-valued bits in the bitstream just prior to the gob_resync_marker in order to obtain byte alignment. The gob_resync_marker shall not be present for the first GOB (for which gob_number = 0).

gob_number: This is a five-bit number which indicates which GOB is being processed in the vop. Its value may either be read following a gob_resync_marker or may be inferred from the progress of macroblock decoding. All GOBs shall appear in the bitstream of each video_plane_with_short_header(), and the GOBs shall appear in a strictly increasing order in the bitstream. In other words, if a gob_number is read from the bitstream after a gob_resync_marker, its value must be the same as the value that would have been inferred in the absence of the gob_resync_marker.

gob_frame_id: This is a two bit field which is intended to help determine whether the data following a gob_resync_marker can be used in cases for which the vop header of the video_plane_with_short_header() may have been lost. gob_frame_id shall have the same value in every GOB header of a given video_plane_with_short_header(). Moreover, if any field among the split_screen_indicator or document_camera_indicator or full_picture_freeze_release or source_format or picture_coding_type as indicated in the header of a video_plane_with_short_header() is the same as for the previous transmitted picture in the same video object, gob_frame_id shall have the same value as in that previous video_plane_with_short_header(). However, if any of these fields in the header of a certain video_plane_with_short_header() differs from that in the previous transmitted video_plane_with_short_header() of the same video object, the value for gob_frame_id in that picture shall differ from the value in the previous picture.

num_macroblocks_in_gob: This is the number of macroblocks in each group of blocks (GOB) unit. This parameter is derived from the source_format as shown in Table 6-25. The count of stuffing macroblocks is not included in this value.

short_video_end_marker: This is a 22-bit end of sequence marker containing the value '0000 0000 0000 0000 1111 11'. It is used to mark the end of a sequence of video_plane_with_short_header(). short_video_end_marker may (and should) be byte aligned by the insertion of zero to seven zero-valued bits to achieve byte alignment prior to short_video_end_marker.

6.3.5.3 Shape coding

bab_type: This is a variable length code between 1 and 7 bits. It indicates the coding mode used for the bab. There are seven bab_types as depicted in Table 6-26 . The VLC tables used depend on the decoding context i.e. the bab_types of blocks already received. For I-VOPs, the context-switched VLC table of Table B-27 is used. For P-VOPs, B-VOPs, and S(GMC)-VOPs, the context switched table of Table B-28 is used.

Table 6-26 -- List of bab_types and usage

bab_type	Semantic	Used in
0	MVDs==0 && No Update	P,B, and S(GMC)- VOPs
1	MVDs!=0 && No Update	P,B, and S(GMC)-VOPs
2	transparent	All VOP types
3	opaque	All VOP types
4	intraCAE	All VOP types
5	MVDs==0 && interCAE	P,B, and S(GMC)-VOPs

6	MVDs!=0 && interCAE	P,B, and S(GMC)-VOPs	

The bab_type determines what other information fields will be present for the bab shape. No further shape information is present if the bab_type = 0, 2 or 3. Opaque means that all pixels of the bab are part of the object. Transparent means that none of the bab pixels belong to the object. IntraCAE means the intra-mode CAE decoding will be required to reconstruct the pixels of the bab. No_update means that motion compensation is used to copy the bab from the previous VOP' s binary alpha map. InterCAE means the motion compensation and inter_mode CAE decoding are used to reconstruct the bab. MVDs refers to the motion vector difference for shape.

mvds_x: This is a VLC code between 1 and 18 bits. It represents the horizontal element of the motion vector difference for the bab. The motion vector difference is in full integer precision. The VLC table is shown is Table B-29.

mvds_y: This is a VLC code between 1 and 18 bits. It represents the vertical element of the motion vector difference for the bab. The motion vector difference is in full integer precision. If mvds_x is '0', then the VLC table of Table B-30 , otherwise the VLC table of Table B-29 is used.

conv_ratio: This is VLC code of length 1-2 bits. It specifies the factor used for sub-sampling the 16x16 pixel bab. The decoder must up-sample the decoded bab by this factor. The possible values for this factor are 1, 2 and 4 and the VLC table used is given in Table B-31.

scan_type: This is a 1-bit flag where a value of '0' implies that the bab is in transposed form i.e. the BAB has been transposed prior to coding. The decoder must then transpose the bab back to its original form following decoding. If this flag is '1', then no transposition is performed.

binary_arithmetic_code(): This is a binary arithmetic decoder representing the pixel values of the bab. This code may be generated by intra cae or inter cae depending on the bab_type. Cae decoding relies on the knowledge of intra_prob[] and inter_prob[], probability tables given in annex B.

enh_bab_type: This is a variable length code between 1 and 3 bits. It indicates the bab coding mode used in binary shape enhancement layer coding. There are four enh_bab_types as depicted in Table V2 - 3. The VLC tables used depend on the decoding context i.e. the bab_types of blocks in lower reference layer.

Table V2 - 3 – List of enh_bab_types and usage

enh_bab_type	Semantic	Used in
0	intra NOT_CODED	P-, B-VOPs
1	intra CODED	P-, B-VOPs
2	inter NOT_CODED	B- VOPs
3	inter CODED	B- VOPs

The enh_bab_type determines what other information fields will be present for the bab shape. No further shape information is present if the enh_bab_type = 0 or 2. NOT_CODED means that motion compensation is used to copy the bab from the reference bab's binary alpha map. In intra NOT_CODED mode, the upsampled bab from the collocated block in lower reference layer is used for the reference bab. In inter NOT_CODED mode, motion compensated bab in the previous VOP of the current layer is used for the reference bab. And the motion vector of the collocated block in the lower reference layer is used for the motion compensation. Each component of the lower layer shape motion vectors are scaled by the up-sampling ratio according to subclause 7.5.4.5 for referencing in the enhancement layer. Intra CODED means Scan Interleaving(SI) based CAE decoding will be required to reconstruct the pixels of the current bab. Inter CODED means the motion compensation and inter_mode CAE decoding are used to reconstruct the current bab.

enh_binary_arithmetic_code() – This is a binary arithmetic decoder representing the pixel values of the bab and bab type of Scan Interleaving (SI) method. This code may be generated by SI decoding method or inter cae depending on the enh_bab_type. When enh_bab_type==1, the first decoded value represents the bab type of SI decoding method ("0": transitional bab, "1": exceptional bab). And the other decoded values represent the pixel values of the bab. If the bab type of SI is "transitional bab" , only transitional pixels in the coded-scan-lines are decoded. Otherwise, for "exceptional bab" , all of the pixels in the coded-scan-lines are decoded. This binary value

decoding relies on the knowledge of enh_bab_type_prob[], and enh_intra_v_prob[] and enh_intra_h_prob[]probability tables given in Annex B. When enh_bab_type==3, this binary arithmetic decoder represents the pixel values of the bab. This code is generated by inter cae. This binary value decoding relies on the knowledge of inter_prob[], probability tables given in Annex B.

6.3.5.4 Sprite coding

warping_mv_code(dmv) : The codeword for each differential motion vector consists of a VLC indicating the length of the dmv code (dmv_length) and a FLC, dmv_code-, with dmv_length bits. The codewords are listed in Table B-33.

brightness_change_factor (): The codeword for brightness_change_factor consists of a variable length code denoting brightness_change_factor_size and a fix length code, brightness_change_factor, of brightness_change_factor_size bits (sign bit included). The codewords are listed in Table B-34.

send_mb(): This function returns 1 if the current macroblock has already been transmitted. Otherwise it returns 0.

piece_quant: This is a 5-bit unsigned interger which indicates the quant to be used for a sprite-piece until updated by a subsequent dquant. The piece_quant carries the binary representation of quantizer values from 1 to 31 in steps of 1.

piece_width: This value specifies the width of the sprite piece measured in macroblock units.

piece_height: This value specifies the height of the sprite piece measured in macroblock units.

piece_xoffset This value specifies the horizontal offset location, measured in macroblock units from the left edge of the sprite object, for the placement of the sprite piece into the sprite object buffer at the decoder.

piece_yoffset: This value specifies the vertical offset location, measured in macroblock units from the top edge of the sprite object.

decode_sprite_piece (): It decodes a selected region of the sprite object or its update. It also decodes the parameters required by the decoder to properly incorporate the pieces. All the static-sprite-object pieces will be encoded using a subset of the I-VOP syntax. And the static-sprite-update pieces use a subset of the P-VOP syntax. The sprite update is defined as the difference between the original sprite texture and the reconstructed sprite assembled from all the sprite object pieces.

sprite_shape_texture(): For the static-sprite-object pieces, shape and texture are coded using the macroblock layer structure in I-VOPs. And the static-sprite-update pieces use the P-VOP inter-macroblock syntax -- except that there are no motion vectors and shape information included in this syntax structure. Macroblocks raster scanning is employed to encode a sprite piece; however, whenever the scan encounters a macroblock which has been part of some previously sent sprite piece, then the block is not coded and the corresponding macroblock layer is empty.

6.3.6 Macroblock related

not_coded: This is a 1-bit flag which signals if a macroblock is coded or not. When set to'1' it indicates that a macroblock is not coded and no further data is included in the bitstream for this macroblock (with the exception of alpha data that may be present). The decoder shall treat this macroblock as 'inter' with motion vector equal to zero and no DCT coefficient data for P-VOPs, and the decoder shall treat this macroblock as 'GMC macroblock (i.e. prediction using the global motion compensated image)' with no motion vector data and no DCT coefficient data for S(GMC)-VOPs. When set to '0' it indicates that the macroblock is coded and its data is included in the bitstream.

mcbpc: This is a variable length code that is used to derive the macroblock type and the coded block pattern for chrominance . It is always included for coded macroblocks. Table B-6 and Table B-7 list all allowed codes for mcbpc in I-, P-, and S(GMC)- VOPs respectively. The values of the column " MB type" in these tables are used as the variable "derived_mb_type" which is used in the respective syntax part for motion and texture decoding. In P-

vops using the short video header format (i.e., when short_video_header is 1), mcbpc codes indicating macroblock type 2 shall not be used.

mcsel: This is a 1-bit flag that specifies the reference image of each macroblock in S-VOPs. This flag is present only when sprite_enable == "GMC," vop_coding_type == "S", and the macroblock type specified by mcbpc is " inter" or "inter+q". mcsel indicates whether the global motion compensated image or the previous reconstructed VOP is referred to for interframe prediction. This flag is set to '1' when GMC is used for the macroblock, and is set to '0' if local MC is used. If mcsel = "1", local motion vectors are not transmitted. The default value for mcsel is '1' (i.e. prediction using GMC) when sprite_enable == " GMC," vop_coding_type == "S", and not_coded == '1'.

ac_pred_flag: This is a 1-bit flag which when set to '1' indicates that either the first row or the first column of ac coefficients are differentially coded for intra coded macroblocks.

cbpy: This variable length code represents a pattern of non-transparent luminance blocks with at least one non intra DC transform coefficient, in a macroblock. Table B-8 – Table B-11 indicate the codes and the corresponding patterns they indicate for the respective cases of intra- and inter-MBs.

dquant: This is a 2-bit code which specifies the change in the quantizer, quant, for I-, P-, and S(GMC)-VOPs. Table 6-27 lists the codes and the differential values they represent. The value of quant lies in range of 1 to $2^{quant\_precision}$-1; if the value of quant after adding dquant value is less than 1 or exceeds $2^{quant\_precision}$-1, it shall be correspondingly clipped to 1 and $2^{quant\_precision}$-1. If quant_precision takes its default value of 5, the range of allowed values for quant is [1:31].

Table 6-27 -- dquant codes and corresponding values

dquant code	value
00	-1
01	-2
10	1
11	2

co_located_not_coded: The value of this internal flag is set to 1 when the current VOP is a B-VOP, the future reference VOP is a P-VOP, and the co-located macroblock in the future reference VOP is skipped (i.e. coded as not_coded = '1'). Otherwise the value of this flag is set to 0. The co-located macroblock is the macroblock which has the same horizontal and vertical index with the current macroblock in the B-VOP. If the co-located macroblock lies outside of the bounding rectangle, this macroblock is considered to be not skipped.

modb: This is a variable length code present only in coded macroblocks of B-VOPs. It indicates whether mb_type and/or cbpb information is present for a macroblock. The codes for modb are listed in Table B-3.

mb_type: This variable length code is present only in coded macroblocks of B-VOPs. Further, it is present only in those macroblocks for which one motion vector is included. The codes for mb_type are shown in Table B-4 for B-VOPs for no scalability and in Table B-5 for B-VOPs with scalability. When mb_type is not present (i.e. modb=='1') for a macroblock in a B-VOP, the macroblock type is set to the default type. The default macroblock type for the enhancement layer of spatially scalable bitstreams (i.e. ref_select_code == '00' && scalability = '1') is "forward mc + Q". Otherwise, the default macroblock type is "direct".

cbpb: This is a 3 to 6 bit code representing coded block pattern in B-VOPs, if indicated by modb. Each bit in the code represents a coded/no coded status of a block; the leftmost bit corresponds to the top left block in the macroblock. For each non-transparent blocks with coefficients, the corresponding bit in the code is set to '1'. When cbpb is not present (i.e. modb=='1' or '01') for a macroblock in a B-VOP, no coefficients are coded for all the non-transparent blocks in this macroblock.

dbquant: This is a variable length code which specifies the change in quantizer for B-VOPs. Table 6-28 lists the codes and the differential values they represent. If the value of quant after adding dbquant value is less than 1 or exceeds $2^{quant\_precision}$-1, it shall be correspondingly clipped to 1 and $2^{quant\_precision}$-1. If quant_precision takes its default value of 5, the range of allowed values for the quantzer for B-VOPs is [1:31].

Table 6-28 -- dbquant codes and corresponding values

dbquant code	value
10	-2
0	0
11	2

coda_i: This is a one-bit flag which is set to "1" to indicate that all the values in the grayscale alpha macroblock are equal to 255 (AlphaOpaqueValue). When set to "0", this flag indicates that one or more 8x8 blocks are coded according to cbpa.

ac_pred_flag_alpha: This is a one-bit flag which when set to '1' indicates that either the first row or the first column of ac coefficients are to be differentially decoded for intra alpha macroblocks. It has the same effect for alpha as the corresponding luminance flag.

cbpa: This is the coded block pattern for grayscale alpha texture data. For I, P, S(GMC), and B VOPs, this VLC is exactly the same as the INTER (P or S(GMC)) cbpy VLC described in Table B-8 – Table B-11. cbpa is followed by the alpha block data which is coded in the same way as texture block data. Note that grayscale alpha blocks with alpha all equal to zero (transparent) are not included in the bitstream.

coda_pb: This is a VLC indicating the coding status for P, S(GMC), or B alpha macroblocks. The semantics are given in the table below (Table 6-29). When this VLC indicates that the alpha macroblock is all opaque, this means that all values are set to 255 (AlphaOpaqueValue).

Table 6-29 -- coda_pb codes and corresponding values

coda_pb	Meaning
1	alpha residue all zero
01	alpha macroblock all opaque
00	alpha residue coded

6.3.6.1 MB Binary Shape Coding

bab_type: This defines the coding type of the current bab according to Table B-27 and Table B-28 for intra and inter mode, respectively.

mvds_x: This defines the size of the x-component of the differential motion vector for the current bab according to Table B-29.

mvds_y: This defines the size of the y-component of the differential motion vector for the current bab according to Table B-29 if mvds_x!=0 and according to Table B-30 if mvds_x==0.

conv_ratio: This defines the upsampling factor according to Table B-31 to be applied after decoding the current shape information

scan_type: This defines according to Table 6-30 whether the current bordered to be decoded bab and the eventual bordered motion compensated bab need to be transposed

Table 6-30 -- scan_type

scan_type	meaning
0	transpose bab as in matrix transpose

1	do not transpose

binary_arithmetic_code() –This is a binary arithmetic decoder that defines the context dependent arithmetically to be decoded binary shape information. The meaning of the bits is defined by the arithmetic decoder according to subclause 7.5 .3

enh_bab_type -- This defines the coding type of the current bab in the enhancement layer according to Table V2 - 30 and Table V2 - 31 for P-VOP and B-VOP coding, respectively.

enh_binary_arithmetic_code() – This is a binary arithmetic decoder that defines the context dependent arithmetically to be decoded binary shape information in the enhancement layer. The meaning of the bits is defined by the arithmetic decoder according to subclause 7.5.3.

6.3.6.2 Motion vector

horizontal_mv_data: This is a variable length code, as defined in Table B-12, which is used in motion vector decoding as described in subclause 7.6.3.

vertical_mv_data: This is a variable length code, as defined in Table B-12, which is used in motion vector decoding as described in subclause 7.6.3.

horizontal_mv_residual: This is an unsigned integer which is used in motion vector decoding as described in subclause 7.6.3. The number of bits in the bitstream for horizontal_mv_residual, r_size, is derived from either vop_fcode_forward or vop_fcode_backward as follows;

r_size = vop_fcode_forward - 1 or r_size = vop_fcode_backward - 1

vertical_mv_residual: This is an unsigned integer which is used in motion vector decoding as described in subclause 7.6.3. The number of bits in the bitstream for vertical_mv_residual, r_size, is derived from either vop_fcode_forward or vop_fcode_backward as follows;

r_size = vop_fcode_forward - 1 or r_size = vop_fcode_backward - 1

6.3.6.3 Interlaced Information

dct_type: This is a 1-bit flag indicating whether the macroblock is frame DCT coded or field DCT coded. If this flag is set to "1", the macroblock is field DCT coded; otherwise, the macroblock is frame DCT coded. This flag is only present in the bitstream if the interlaced flag is set to "1" and the macroblock is coded (coded blcok pattern is non-zero) or intra-coded. Boundary blocks are always coded in frame-based mode.

field_prediction: This is a 1-bit flag indicating whether the macroblock is field predicted or frame predicted. This flag is set to '1' when the macroblock is predicted using field motion vectors. If it is set to '0' then frame prediction (16x16 or 8x8) will be used. This flag is only present when interlaced == '1' for the following types of macrloblocks: a macroblock in a P-VOP with derived_mb_type < 2; a non-direct mode macrloblock in a B-VOP; or a macroblocks in an S (GMC)-VOP with mcsel == '0'.

forward_top_field_reference: This is a 1-bit flag which indicates the reference field for the forward motion compensation of the top field. When this flag is set to '0', the top field is used as the reference field. If it is set to '1' then the bottom field will be used as the reference field. This flag is only present in the bitstream if the field_prediction flag is set to "1" and the macroblock is not backward predicted.

forward_bottom_field_reference: This is a 1-bit flag which indicates the reference field for the forward motion compensation of the bottom field. When this flag is set to '0', the top field is used as the reference field. If it is set to '1' then the bottom field will be used as the reference field. This flag is only present in the bitstream if the field_prediction flag is set to "1" and the macroblock is not backward predicted.

backward_top_field_reference: This is a 1-bit flag which indicates the reference field for the backward motion compensation of the top field. When this flag is set to '0', the top field is used as the reference field. If it is set to '1'

then the bottom field will be used as the reference field. This flag is only present in the bitstream if the field_prediction flag is set to "1" and the macroblock is not forward predicted.

backward_bottom_field_reference: This is a 1-bit flag which indicates the reference field for the backward motion compensation of the bottom field. When this flag is set to '0', the top field is used as the reference field. If it is set to '1' then the bottom field will be used as the reference field.. This flag is only present in the bitstream if the field_prediction flag is set to "1" and the macroblock is not forward predicted.

6.3.7 Block related

intra_dc_coefficient: This is a fixed length code that defines the value of an intra DC coefficient when the short video header format is in use (i.e., when short_video_header is "1"). It is transmitted as a fixed length unsigned integer code of size 8 bits, unless this integer has the value 255. The values 0 and 128 shall not be used – they are reserved. If the integer value is 255, this is interpreted as a signalled value of 128. The integer value is then multiplied by a dc_scaler value of 8 to produce the reconstructed intra DC coefficient value.

dct_dc_size_luminance: This is a variable length code as defined in Table B-13 that is used to derive the value of the differential dc coefficients of luminance values in blocks in intra macroblocks. This value categorizes the coefficients according to their size.

dct_dc_differential: This is a variable length code as defined in Table B-15 that is used to derive the value of the differential dc coefficients in blocks in intra macroblocks. After identifying the category of the dc coefficient in size from dct_dc_size_luminance or dct_dc_size_chrominance, this value denotes which actual difference in that category occurred.

dct_dc_size_chrominance: This is a variable length code as defined in Table B-14 that is used to derive the value of the differential dc coefficients of chrominance values in blocks in intra macroblocks. This value categorizes the coefficients according to their size.

pattern_code[i]: The value of this internal flag is set to 1 if the block or alpha block with the index value i includes one or more DCT coefficients that are decoded using at least one of Table B-16 to Table B-25. Otherwise the value of this flag is set to 0.

6.3.7.1 Alpha block related

dct_dc_size_alpha: This is a variable length code for coding the alpha block dc coefficient. Its semantics are the same as dct_dc_size_luminance in subclause 6.3.7.

6.3.8 Still texture object

still_texture_object_start_code: The still_texture_object_start_code is a string of 32 bits. The first 24 bits are '0000 0000 0000 0000 0000 0001' and the last 8 bits are defined in Table 6-3.

texture_tile_start_code: The texture_tile_start_code is a string of 32 bits. The 32 bits are '0000 0000 0000 0000 0000 0001 1100 0001' in binary. The texture_tile_start code marks the start of a new tile.

tiling_disable: This field indicates the succeeding bitstream contains a structure of tile when the field is '0'.

tile_width: This is a 15-bit unsigned integer which specifies horizontal size, in pexel unit, of the rectangle. When texture_object_layer_shape==' 00' , this value must be lower than texture_object_layer_width and a zero value is forbidden. When texture_object_layer_width is not a multilple of tile_width, the horizontal size of tile in the last column is derived by texture_object_layer_width%tile_width. When texture_object_layer_shape=='01', object_width is used instead of texture_object_layer_width. The value of tile_width shall be divisible by two.

tile_height: This is a 15-bit unsigned integer which specifies vertical size, in pexel unit, of the rectangle. When texture_object_layer_shape==' 00' , this value must be lower than texture_object_layer_height and a zero value is forbidden. When texture_object_layer_height is not a multilple of tile_height, the vertical size of tile in the last row is

derived by texture_object_layer_height%tile_height. When texture_object_layer_shape== '01', object_height is used instead of texture_object_layer_height. The value of tile_height shall be divisible by two.

number_of_tiles: This is a 16-bit of unsigned integer specifying the number of tiles encoded in this bitstream. When texture_object_layer_shape=='00', the value is derived from CEIL(texture_object_layer_width÷tile_width) * CEIL(texture_object_layer_height÷tile_height), where CEIL() rounds up to the nearest integer. When texture_object_layer_shape==' 01' , the value is derived from CEIL(object_width÷tile_width) * CEIL(object_height÷tile_height).

tiling_jump_table_enable: This field indicates the succeeding bitstream contains a size of bitstream for each tile when the field is '1'.

tile_size_high: This is a left part of 16-bit of a unsigned integer in 32-bit which indicates a size of bitstream containing the corresponding tile in byte unit.

tile_size_low: This is a right part of 16-bit of a unsigned integer in 32-bit which indicates a size of bitstream containing the corresponding tile in byte unit. The real size of bitstream for a certain tile is derived from 'tile_size_high<<16+tile_size_low'.

tile_id: This is given by 16-bits representing one of the values in the range of ' 0000' to 'FFFF' in hexadecimal starting from top-left ended to bottom-right. The field uniquely identifies each tile.

texture_error_resilience_disable: This is a one-bit flag which when set to '0' indicates that the Still Texture Object is operating in error resilience mode.

target_segment_length: This parameter specifies the minimum number of bits in a segment within a packet before adding a segment marker.

decode_segment_marker(): This function will decode the arithmetically encoded segment marker. The arithmetic model used is the initial model used in decoding type information for the color in which the segment marker is. When in error free case, a ZTR symbol will be decoded as the segment marker.

texture_object_id: This is given by 16-bits representing one of the values in the range of ' 0000 0000 0000 0000' to '1111 1111 1111 1111' in binary. The texture_object_id uniquely identifies a texture object layer.

wavelet_filter_type: This field indicates the arithmetic precision which is used for the wavelet decomposition as the following:

Table 6-31 -- Wavelet type

wavelet_filter_type	Meaning
0	integer
1	Double float

wavelet_download: This field indicates if the 2-band filter bank is specificed in the bitstream:

Table 6-32 -- Wavelet downloading flag

wavelet_download	meaning
0	default filters
1	specified in bitstream

The default filter banks are described in subclause B.2.2.

wavelet_decomposition_levels: This field indicates the number of levels in the wavelet decomposition of the texture.

scan_direction: This field indicates the scan order of AC coefficients. In single-quant and multi-quant mode, if this flag is `0', then the coefficients are scanned in the tree-depth fashion. If it is `1', then they are scanned in the subband by subband fashion. In bilevel_quant mode, if the flag is `0', then they are scanned in bitplane by bitplane fashion. Within each bitplane, they are scanned in a subband by subband fashion. If it is "1", they are scanned from the low wavelet decomposition layer to high wavelet decomposition layer. Within each wavelet decomposition layer, they are scanned from most significant bitplane down to the least significant bitplane.

start_code_enable: If this flag is enabled (disable =0; enabled = 1), the start code followed by an ID to be inserted in to each spatial scalability layer and/or each SNR scalability layer.

texture_object_layer_shape: This is a 2-bit integer defined in Table 6-33. It identifies the shape type of a texture object layer.

Table 6-33 -- Texture Object Layer Shape type

texture_object_layer_shape	Meaning
0 0	rectangular
0 1	binary
1 0	reserved
1 1	reserved

quantisation_type: This field indicates the type of quantisation as shown in Table 6-34.

Table 6-34 -- The quantisation type

quantisation_type	Code
single quantizer	0 1
multi quantizer	1 0
bi-level quantizer	1 1

spatial_scalability_levels: This field indicates the number of spatial scalability layers supported in the bitstream. This number can be from 1 to wavelet_decomposition_levels.

use_default_spatial_scalability: This field indicates how the spatial scalability levels are formed. If its value is one, then default spatial scalability is used, starting from (¼)^(spatial_scalability_levels1)-th of the full resolution up to the full resolution, where ^ is a power operation. If its value is zero, the spatial scalability is specified by wavelet_layer_index described below.

wavelet_layer_index: This field indicates the identification number of wavelet_decomposition layer used for spatial scalability. The index starts with 0 (i.e., root_band) and ends at (wavelet_decomposition_levels–1) (i.e., full resolution).

uniform_wavelet_filter: If this field is "1" , then the same wavelet filter is applied for all wavelet layers. If this field is "0", then different wavelet filters may be applied for the wavelet decomposition. Note that the same filters are used for both luminance and chromanence. Since the chromanence' s width and height is half that of the luminance, the last wavelet filter applied to the luminance is skipped when the chromanence is synthesized.

wavelet_stuffing: These 3 stuffing bits are reserved for future expansion. It is currently defined to be '1 1 1'.

texture_object_layer_width: The texture_object_layer_width is a 15-bit unsigned integer representing the width of the displayable part of the luminance component in pixel units. A zero value is forbidden.

texture_object_layer_height: The texture_object_layer_height is a 15-bit unsigned integer representing the height of the displayable part of the luminance component in pixel units. A zero value is forbidden.

horizontal_ref: This is a 15-bit integer which specifies, in pixel units, the horizontal position of the top left of the rectangle defined by horizontal size of object_width. The value of horizontal_ref shall be divisible by two. This is used for decoding and for picture composition.

vertical_ref: This is a 15-bit integer which specifies, in pixel units, the vertical position of the top left of the rectangle defined by vertical size of object_height. The value of vertical_ref shall be divisible by two. This is used for decoding and for picture composition.

object_width: This is a 15-bit unsigned integer which specifies the horizontal size, in pixel units, of the rectangle that includes the object. A zero value is forbidden.

object_height: This is a 15-bit unsigned integer which specifies the vertical size, in pixel units, of the rectangle that includes the object. A zero value is forbidden.

quant_byte: This field defines one byte of the quantisation step size for each scalability layer. A zero value is forbidden. The quantisation step size parameter, quant, is decoded using the function get_param( ): quant = get_param(7);

max_bitplanes: This field indicates the number of maximum bitplanes in all three quantization modes.

texture_tile_type: This is a 2-bit integer defined in Table V2 - 4. It identifies the shape type of a texture object in a tile when texture_object_layer_shape is " 01".

Table V2 - 4 -- Texture Tile Shape

texture_tile_type	Meaning
00	Forbiden
01	opaque tile
10	Boundary tile
11	Transparent tile

next_texture_marker ():This function performs a similar operation as next_start_code(), but for texture_marker.

texture_marker – This is a binary string of at least 16 zero' s followed by a one '0 0000 0000 0000 0001'. It is only present when texture_error_resilience_disable_flag is set to '0' . A texture marker shall only be located immediately before a texture packet and aligned with a byte.

TU_first – This parameter specifies the number of the first texture unit within the texture packet. This parameter is decoded by the function get_param().

TU_last – This parameter specifies the number of the last texture unit within the texture packet. This parameter is decoded by the function get_param().

header_extention_code – This is a one-bit flag which when set to '1' indicates that additional header information is sent in the texture packet. This bit must have value 1 in the first packet of a texture object.

target_segment_length – This parameter specifies the minimum number of bits in a segment within a packet before adding a segment marker.

6.3.8.1 Texture Layer Decoding

arith_decode_highbands_td(): This is an arithmetic decoder for decoding the quantized coefficient values of the higher bands (all bands except DC band) within a single tree block. The bitstream is generated by an adaptive arithmetic encoder. The arithmetic decoding relies on the initialization of the uniform probability distribution models described in subclause B.2.2. This decoder uses only integer arithmetic. It also uses an adaptive probability model based on the frequency counts of the previously decoded symbols. The maximum range (or precision) specified is (2^16) - 1 (16 bits). The maximum frequency count for the magnitude and residual models is 127, and for all other models it is 127. The arithmetic coder used is identical to the one used in arith_decode_highbands_bilevel_td().

texture_spatial_layer_start_code: The texture_spatial_layer_start_code is a string of 32 bits. The 32 bits are '0000 0000 0000 0000 0000 0001 1011 1111' in binary. The texture_spatial_layer_start_code marks the start of a new spatial layer.

texture_spatial_layer_id: This is given by 5-bits representing one of the values in the range of '00000' to '11111' in binary. The texture_spatial_layer_id uniquely identifies a spatial layer.

arith_decode_highbands_bb(): This is an arithmetic decoder for decoding the quantized coefficient values of the higher bands (all bands except DC band) within a single band. The bitstream is generated by an adaptive arithmetic encoder. The arithmetic decoding relies on the initialization of the uniform probability distribution models described in subclause B.2.2. This decoder uses arithmetic. It also uses an adaptive probability model based on the frequency counts of the previously decoded symbols. The maximum range (or precision) specified is (2^16) - 1 (16 bits). The maximum frequency count for the magnitude and residual models is 127, and for all other models it is 127.

snr_scalability_levels: This field indicates the number of levels of SNR scalability supported in this spatial scalability level.

texture_snr_layer_start_code: The texture_snr_layer_start_code is a string of 32 bits. The 32 bits are '0000 0000 0000 0000 0000 0001 1100 0000' in binary. The texture_snr_layer_start_code marks the start of a new snr layer.

texture_snr_layer_id: This is given by 5-bits representing one of the values in the range of '00000' to '11111' in binary. The texture_snr_layer_id uniquely identifies an SNR layer.

NOTE All the start codes start at the byte boundary. Appropriate number of bits is stuffed before any start code to byte-align the bitstream.

all_nonzero: This flag indicates whether some of the subbands of the current layer contain only zero coefficients. The value '0' for this flag indicates that one or more of the subbands contain only zero coefficients. The value '1' for this flag indicates the all the subbands contain some nonzero coefficients

all_zero: This flag indicates whether all the coefficients in the current layer are zero or not. The value '0' for this flag indicates that the layer contains some nonzero coefficients. The value '1' for this flag indicates that the layer only contains zero coefficients, and therefore the layer is skipped.

lh_zero, hl_zero, hh_zero: This flag indicates whether the LH/HL/HH subband of the current layer contains only all zero coefficients. The value '1' for this flag indicates that the LH/HL/HH subband contains only zero coefficients, and therefore the subband is skipped. The value '0' for this flag indicates that the LH/HL/HH subband contains some nonzero coefficients

arith_decode_highbands_bilevel_bb(): This is an arithmetic decoder for decoding the quantized coefficient values of the higher bands in the bilevel_quant mode (all bands except DC band). The bitstream is generated by an adaptive arithmetic encoder. The arithmetic decoding relies on the initialization of the uniform probability distribution models described. The arith_decode_highbands_bilevel() function uses bitplane scanning, and a different probability model as described in subclause B.2.2. In this mode, The maximum range (or precision) specified is (2^16) - 1 (16 bits). The maximum frequency count is 127. It uses the lh/hl/hh_zero flags to see if any of the LH/HL/HH are all zero thus not decoded. For example if lh_zero=1 and hh_zero=1 only hl_zero is decoded.

arith_decode_highbands_bilevel_td(): This is an arithmetic decoder for decoding the quantized coefficient values of the higher bands in the bilevel_quant mode (all bands except DC band). The bitstream is generated by an adaptive arithmetic encoder. The arithmetic decoding relies on the initialization of the uniform probability distribution models described. The arith_decode_highbands_bilevel() function uses bitplane scanning, and a different probability model as described in subclause B.2.2. In this mode, The maximum range (or precision) specified is (2^16) - 1 (16 bits). The maximum frequency count is 127. It uses the lh/hl/ll_zero flags to see if any of the LH/HL/HH are all zero thus not decoded. For example if lh_zero=1 and hh_zero=1 only hl_zero is decoded.

lowpass_filter_length: This field defines the length of the low pass filter in binary ranging from "0001" (length of 1) to " 1111" (length of 15.)

highpass_filter_length: This field defines the length of the high pass filter in binary ranging from "0001" (length of 1) to " 1111" (length of 15.)

filter_tap_integer: This field defines an integer filter coefficient in a 16 bit signed integer. The filter coefficients are decoded from the left most tap to the right most tap order.

filter_tap_float_high: This field defines the left 16 bits of a floating filter coefficient which is defined in 32-bit IEEE floating format. The filter coefficients are decoded from the left most tap to the right most tap order.

filter_tap_float_low: This field defines the right 16 bits of a floating filter coefficient which is defined in 32-bit IEEE floating format. The filter coefficients are decoded from the left most tap to the right most tap order.

integer_scale: This field defines the scaling factor of the integer wavelet, by which the output of each composition level is divided by an integer division operation. A zero value is forbidden.

mean: This field indicates the mean value of one color component of the texture.

quant_dc_byte: This field indicates the quantization step size for one color component of the DC subband. A zero value is forbidden. The quantization step size parameter, quant_dc, is decoded using the function get_param( ): quant_dc = get_param(7); where get_param() function is defined in the description of band_offset_byte.

band_offset_byte: This field defines one byte of the absolute value of the parameter band_offset. This parameter is added to each DC band coefficient obtained by arithmetic decoding. The parameter band_offset is decoded using the function get_param( ):

```
band_offset = -get_param(7);
```

where function get_param() is defined as

```
int get_param(int nbit)
{
int count = 0;
int word =0;
int value = 0;
int module = 1<<(nbit);

    do{
word= get_next_word_from_bitstream( nbit+1);
value += (word & (module-1) ) << (count * nbit);
count ++;
} while( word>> nbit);
    return value;
}
```

The function get_next_word_from_bitstream(x) reads the next x bits from the input bitstream.

band_max_byte: This field defines one byte of the maximum value of the DC band. The parameter band_max_value is decoded using function get_param(). The number of maximum bitplanes for DC band is derived from CEIL($\log_2$(band_max_value+1))

```
band_max_value = get_param(7);
```

arith_decode_dc(): This is an arithmetic decoder for decoding the quantized coefficient values of DC band only. No zerotree symbol is decoded since the VAL is assumed for all DC coefficient values. This bitstream is generated by an adaptive arithmetic encoder. The arithmetic decoding relies on the initialization of a uniform probability distribution model described in subclause B.2.2. The arith_decode_dc() function uses the same arithmetic decoder as described in arith_decode_highbands_td() but it uses different scanning, and a different probability model (*DC*).

6.3.8.2 Shape Object decoding

change_conv_ratio_disable: This specifies whether conv_ratio is encoded at the shape object decoding function. If it is set to "1" when disable.

sto_constant_alpha: This is a 1-bit flag when set to '1', the opaque alpha values of the binary mask are replaced with the alpha value specified by sto_constant_alpha_value.

sto_constant_alpha_value: This is an 8-bit code that gives the alpha value to replace the opaque pixels in the binary alpha mask. Value '0' is forbidden.

marker_bit: This is one-bit that shall be set to 1. This bit prevents emulation of start codes.

sto_shape_coded_layers: This is a 4-bit unsigned integer to indicate the number of enhancement layers to contained in the bitstream.

texture_shape_layer_start_code: This is a string of 32 bits. The first 24 bits are '0000 0000 0000 0000 0000 0001' and the last 8 bits '1100 0010' (0xC2). This texture_shape_layer_start_code marks the start of a new shape enhancement layer.

texture_shape_layer_id: This is given by a 5-bit number representing one of the values in the range of '00000' to '11111' in binary. The texture_shape_layer_id uniquely identifies a shape spatial layer.

texture_spatial_layer_start_code: This is a string of 32 bits. The first 24 bits are '0000 0000 0000 0000 0000 0001' and the last 8 bits '1011 1111' (0xBF). This texture_spatial_layer_start_code marks the start of texture decoding.

texture_spatial_layer_id: This is given by a 5-bit number representing one of the values in the range of '00000' to '11111' in binary. This texture_spatial_layer_id uniquely identifies the start of texture decoding.

shape_base_layer_height_blocks(): This is a function that returns the number of shape blocks in vertical directions in the base layer. In the case that tiling_disable is '1' object_height will be used for the number. The number is given by ((object_height>>wavelet_decomposition_levels)+15)/16.

If tiling_disable is '0' tiling_height will be used instead of object_height. The number is given by ((tile_height>>wavelet_decomposition_levels) +15)/16.

shape_base_layer_width_blocks(): This is a function that returns the number of shape blocks in horizontal directions in the base layer. In the case that tiling_disable is '1' object_width will be used for the number. The number is given by ((object_width>>wavelet_decomposition_levels) +15)/16.

If tiling_disable is '0' tiling_width will be used instead of object_width. The number is given by ((tile_width>>wavelet_decomposition_levels) +15)/16.

bab_type: This is a variable length code of 1-2 bits. It indicates the coding mode used for the bab. There are three bab_types as depicted in Table 6-35. The VLC tables used depend on the decoding context i.e. the bab_types of blocks already received.

Table 6-35 -- List of bab_types and usage

bab_type	Semantic	code
2	transparent	10
3	opaque	0
4	intraCAE	11

The bab_type determines what other information fields will be present for the bab shape. No further shape information is present if the bab_type = 2 or 3. opaque means that all pixels of the bab are part of the object. transparent means that none of the bab pixels belong to the object. IntraCAE means the intra-mode CAE decoding will be required to reconstruct the pixels of the bab.

conv_ratio: This is VLC code of length 1-2 bits. It specifies the factor used for sub-sampling the 16x16 pixel bab. The decoder must up-sample the decoded bab by this factor. The possible values for this factor are 1, 2 and 4 and the VLC table used is given in Table B-31.

scan_type: This is a 1-bit flag where a value of '0' implies that the bab is in transposed form i.e. the bab has been transposed prior to coding. The decoder must then transpose the bab back to its original form following decoding. If this flag is '1', then no transposition is performed.

binary_arithmetic_decode(): This is a binary arithmetic decoder representing the pixel values of the bab. Cae decoding relies on the knowledge of intra_prob[], probability tables given in annex B.

shape_enhanced_layer_height_blocks(): This is a function that returns the number of shape blocks in vertical directions in the enhancement layer. In the case that tiling_disable is '1', object_height will be used for the number. If the current coding layer is the *L*-th layer in the bitstream, the number of blocks is given by

((object_height>>(wavelet_decomposition_levels–*L*–1))+bab_size–1)/bab_size.

If tiling_disable is '0' tiling_height will be used instead of object_height. If the current coding layer is the *L*-th layer in the bitstream, the number of blocks is given by

((tile_height>>(wavelet_decomposition_levels–*L*–1))+bab_size–1)/bab_size.

The value of bab_size (size of the coded bab) is defined in subclause 7.10.6.2.1.

shape_enhanced_layer_width_blocks(): This is a function that returns the number of shape blocks in horizontal directions in the enhancement layer. In the case that tiling_disable is '1', object_width will be used for the number. If the current coding layer is the *L*-th layer in the bitstream, the number of blocks is given by

((object_width>>(wavelet_decomposition_levels–*L*–1))+bab_size–1)/bab_size.

If tiling_disable is '0' tiling_width will be used instead of object_width. If the current coding layer is the *L*-th layer in the bitstream, the number of blocks is given by

((tile_width>>(wavelet_decomposition_levels–*L*–1))+bab_size–1)/bab_size.

enh_binary_arithmetic_decode(): The first decoded value denotes BAB type of Scan Interleaving (SI) method (SI_bab_type : "0": transitional BAB, "1": exceptional BAB). And the other decoded values represent the pixel values of the current BAB. If the BAB is a transitional BAB, only transitional pixels are decoded. Otherwise all of the pixels are decoded. This binary value decoding relies on the knowledge of SI_bab_type_prob[], enh_intra_v_prob[] and enh_intra_h_prob[], sto_SI_bab_type_prob_even[], sto_enh_odd_prob0[], sto_enh_even_prob0[], sto_enh_odd_prob1[], and sto_enh_even_prob1[] probability tables given in Annex B.

6.3.9 Mesh object

mesh_object_start_code: The mesh_object_start_code is the bit string '000001BC' in hexadecimal. It initiates a mesh object.

6.3.9.1 Mesh object plane

mesh_object_plane_start_code: The mesh_object_plane_start_code is the bit string '000001BD' in hexadecimal. It initiates a mesh object plane.

is_intra: This is a 1-bit flag which when set to '1' indicates that the mesh object is coded in intra mode. When set to '0' it indicates that the mesh object is coded in predictive mode.

6.3.9.2 Mesh geometry

mesh_type_code: This is a 2-bit integer defined in Table 6-36. It indicates the type of initial mesh geometry to be decoded.

Table 6-36 -- Mesh type code

mesh type code	mesh geometry
00	forbidden
01	uniform
10	Delaunay
11	reserved

nr_of_mesh_nodes_hor: This is a 10-bit unsigned integer specifying the number of nodes in one row of a uniform mesh.

nr_of_mesh_nodes_vert: This is a 10-bit unsigned integer specifying the number of nodes in one column of a uniform mesh.

mesh_rect_size_hor: This is a 8-bit unsigned integer specifying the width of a rectangle of a uniform mesh (containing two triangles) in half pixel units.

mesh_rect_size_vert: This is a 8-bit unsigned integer specifying the height of a rectangle of a uniform mesh (containing two triangles) in half pixel units.

triangle_split_code: This is a 2-bit integer defined in Table 6-37. It specifies how rectangles of a uniform mesh are split to form triangles.

Table 6-37 -- Specification of the triangulation type

triangle split code	Split
00	top-left to right bottom
01	bottom-left to top right
10	alternately top-left to bottom-right and bottom-left to top-right
11	alternately bottom-left to top-right and top-left to bottom-right

nr_of_mesh_nodes: This is a 16-bit unsigned integer defining the total number of nodes (vertices) of a (non-uniform) Delaunay mesh. These nodes include both interior nodes as well as boundary nodes.

nr_of_boundary_nodes: This is a 10-bit unsigned integer defining the number of nodes (vertices) on the boundary of a (non-uniform) Delaunay mesh.

node0_x: This is a 13-bit signed integer specifying the x-coordinate of the first boundary node (vertex) of a mesh in half-pixel units with respect to a local coordinate system.

node0_y: This is a 13-bit signed integer specifying the y-coordinate of the first boundary node (vertex) of a mesh in half-pixel units with respect to a local coordinate system.

delta_x_len_vlc: This is a variable-length code specifying the length of the delta_x code that follows. The delta_x_len_vlc and delta_x codes together specify the difference between the x-coordinates of a node (vertex) and the previously encoded node (vertex). The definition of the delta_x_len_vlc and delta_x codes are given in Table B-33, the table for sprite motion trajectory coding.

delta_x: This is an integer that defines the value of the difference between the x-coordinates of a node (vertex) and the previously encoded node (vertex) in half pixel units. The number of bits in the bitstream for delta_x is delta_x_len_vlc.

delta_y_len_vlc: This is a variable-length code specifying the length of the delta_y code that follows. The delta_y_len_vlc and delta_y codes together specify the difference between the y-coordinates of a node (vertex) and the previously encoded node (vertex). The definition of the delta_y_len_vlc and delta_y codes are given in Table B-33, the table for sprite motion trajectory coding.

delta_y: This is an integer that defines the value of the difference between the y-coordinates of a node (vertex) and the previously encoded node (vertex) in half pixel units. The number of bits in the bitstream for delta_y is delta_y_len_vlc.

6.3.9.3 Mesh motion

motion_range_code: This is a 3-bit integer defined in Table 6-38. It specifies the dynamic range of motion vectors in half pel units.

Table 6-38 -- motion range code

motion range code	motion vector range
1	[-32, 31]
2	[-64, 63]
3	[-128, 127]
4	[-256, 255]
5	[-512, 511]
6	[-1024, 1023]
7	[-2048, 2047]

node_motion_vector_flag: This is a 1 bit code specifying whether a node has a zero motion vector. When set to '1' it indicates that a node has a zero motion vector, in which case the motion vector is not encoded. When set to '0', it indicates the node has a nonzero motion vector and that motion vector data shall follow.

delta_mv_x_vlc: This is a variable-length code defining (together with delta_mv_x_res) the value of the difference in the x-component of the motion vector of a node compared to the x-component of a predicting motion vector. The definition of the delta_mv_x_vlc codes are given in Table B-12, the table for motion vector coding (MVD). The value delta_mv_x_vlc is given in half pixel units.

delta_mv_x_res: This is an integer which is used in mesh node motion vector decoding using an algorithm equivalent to that described in the section on video motion vector decoding, subclause 7.6.3. The number of bits in the bitstream for delta_mv_x_res is motion_range_code-1.

delta_mv_y_vlc: This is a variable-length code defining (together with delta_mv_y_res) the value of the difference in the y-component of the motion vector of a node compared to the y-component of a predicting motion vector. The definition of the delta_mv_y_vlc codes are given in Table B-12, the table for motion vector coding (MVD). The value delta_mv_y_vlc is given in half pixel units.

delta_mv_y_res: This is an integer which is used in mesh node motion vector decoding using an algorithm equivalent to that described in the section on video motion vector decoding, subclause 7.6.3. The number of bits in the bitstream for delta_mv_y_res is motion_range_code-1.

6.3.10 FBA object

fba_object_start_code: The fba_object_start_code is the bit string '000001BA' in hexadecimal. It initiates a FBA object.

6.3.10.1 FBA object plane header

fba_object_plane_start_code: The **fba**_frame_start_code is the bit string '000001BB' in hexadecimal. It initiates a FBA object plane.

is_intra: This is a 1-bit flag which when set to '1' indicates that the FBA object is coded in intra mode. When set to '0' it indicates that the FBA object is coded in predictive mode.

fba_object_mask: This is a 2-bit integer defined in Table 6-40. It indicates whether FBA and BAP data are present in the FBA_frame.

Table 6-40 – FBA object mask

mask value	Meaning
00	unused
01	FAP present
10	BAP present
11	both FAP and BAP present

6.3.10.2 FBA object plane data

fap_quant: This is a 5-bit unsigned integer which is the quantization scale factor used to compute the FAPi table step size or DCT fap_scale depending on the fba_object_coding_type. If the fba_object_coding_type is DCT this is a 5-bit unsigned integer used as the index to a fap_scale table for computing the quantization step size of DCT coefficients. The value of fap_scale is specified in the following list:

fap_scale[0 - 31] = { 1, 1, 2, 3, 5, 7, 8, 10, 12, 15, 18, 21, 25, 30, 35, 42,
50, 60, 72, 87, 105, 128, 156, 191, 234, 288, 355, 439, 543, 674, 836, 1039}

fap_mask_type: This is a 2-bit integer. It indicates if the group mask will be present for the specified fap group, or if the complete faps will be present; its meaning is described in Table 6-42. In the case the type is '10' the '0' bit in the group mask indicates interpolate fap.

Table 6-42 -- fap mask type

mask type	Meaning
00	no mask nor fap

01	group mask
10	group mask'
11	fap

fap_group_mask[group_number]: This is a variable length bit entity that indicates, for a particular group_number which fap is represented in the bitstream. The value is interpreted as a mask of 1-bit fields. A 1-bit field in the mask that is set to '1' indicates that the corresponding fap is present in the bitstream. When that 1-bit field is set to '0' it indicates that the fap is not present in the bitstream. The number of bits used for the fap_group_mask depends on the group_number, and is given in Table 6-43.

Table 6-43 -- fap group mask bits

group_number	No. of bits
1	2
2	16
3	12
4	8
5	4
6	5
7	3
8	10
9	4
10	4

NFAP[group_number] : This indicates the number of FAPs in each FAP group. Its values are specified in the following table:

Table 6-44 -- NFAP definition

group_number	NFAP[group_number]
1	2
2	16
3	12
4	8
5	4
6	5
7	3
8	10
9	4
10	4

fba_suggested_gender: This is a 1-bit integer indicating the suggested gender for the face model. It does not bind the decoder to display a facial model of suggested gender, but indicates that the content would be more suitable for display with the facial model of indicated gender, if the decoder can provide one. If fba_suggested_gender is 1, the suggested gender is male, otherwise it is female.

fba_object_coding_type: This is a 1-bit integer indicating which coding method is used. Its meaning is described in Table 6-39.

Table 6-39 -- fba_object_coding_type

type value	Meaning
0	predictive coding
1	DCT

is_i_new_max: This is a 1-bit flag which when set to '1' indicates that a new set of maximum range values for I frame follows these 4, 1-bit fields.

is_i_new_min: This is a 1-bit flag which when set to '1' indicates that a new set of minimum range values for I frame follows these 4, 1-bit fields.

is_p_new_max: This is a 1-bit flag which when set to '1' indicates that a new set of maximum range values for P frame follows these 4, 1-bit fields.

is_p_new_min: This is a 1-bit flag which when set to '1' indicates that a new set of minimum range values for P frame follows these 4, 1-bit fields.

bap_pred_quant_index: This is a 5-bit unsigned integer used as the index to a bap_pred_scale table for computing the quantisation step size of BAP values for predictive and DCT coding. If fba_object_coding_type is predictive, the value of bap_pred_scale is specified in the following list:

bap_pred_scale[0-31]=
{ 0, 1, 2, 3, 5, 7, 9, 11, 14, 17, 20, 23, 27, 31, 35, 39, 43, 47, 52, 57, 62, 67, 72, 77, 82, 88, 94, 100, 106, 113, 120, 127}

If the fba_object_coding_type is DCT this is a 5-bit unsigned integer used as the index to a bap_scale table for computing the quantisation step size of DCT coefficients. The value of bap_scale is specified in the following list:

bap_scale[0 – 31] = { 1, 1, 2, 3, 5, 7, 8, 10, 12, 15, 18, 21, 25, 30, 35, 42,
50, 60, 72, 87, 105, 128, 156, 191, 234, 288, 355, 439, 543, 674, 836, 1039}

bap_mask_type: This 2-bit value determines whether BAPs are transmitted individually or in groups.

bap_mask_type	Meaning
00	No BAPs
01	BAPs transmitted in groups
10	reserved
11	BAPs transmitted individually

bap_group_mask: this is a variable-length mask indicating which BAPs in a group are present in the fba_object_plane.

group number	group name	No. of. bits
1	Pelvis	3
2	Left leg1	4
3	Right leg1	4
4	Left leg2	6
5	Right leg2	6
6	Left arm1	5
7	Right arm1	5
8	Left arm2	7
9	Right arm2	7
10	Spine1	12
11	Spine2	15
12	Spine3	18
13	Spine4	18
14	Spine5	12
15	Left hand1	16
16	Right hand1	16
17	Left hand2	13
18	Right hand2	13
19	Global positioning	6
20	Extension1	22
21	Extension2	22
22	Extension3	22
23	Extension4	22
24	Extension5	22

bap_is_i_new_max: This is a 1-bit flag which when set to '1' indicates that a new set of maximum range values for I frame follows these 4, 1-bit fields.

bap_is_i_new_min: This is a 1-bit flag which when set to '1' indicates that a new set of minimum range values for I frame follows these 4, 1-bit fields.

bap_is_p_new_max: This is a 1-bit flag which when set to '1' indicates that a new set of maximum range values for P frame follows these 4, 1-bit fields.

bap_is_p_new_min: This is a 1-bit flag which when set to '1' indicates that a new set of minimum range values for P frame follows these 4, 1-bit fields.

6.3.10.3 Temporal Header

The frame rate, time code and skip frames information is independently associated with face, body or both depending on the fba_object_mask. For example, if the frame rate is set to 30Hz during a frame with fba_object_mask='01' (face) followed by a frame with frame rate set to 15Hz with fba_object_mask='10' (body) then the frame rate for the face remains set to 30Hz. If the fba_object_mask='11' the temporal information is applied to both face and body.

is_frame_rate: This is a 1-bit flag which when set to '1' indicates that frame rate information follows this bit field. When set to '0' no frame rate information follows this bit field.

is_time_code: This is a 1-bit flag which when set to '1' indicates that time code information follows this bit field. When set to '0' no time code information follows this bit field.

time_code: This is a 18-bit integer containing the following: time_code_hours, time_code_minutes, marker_bit and time_code_seconds as shown in Table 6-41. The parameters correspond to those defined in the IEC standard publication 461 for "time and control codes for video tape recorders". The time code specifies the modulo part (i.e. the full second units) of the time base for the current object plane.

Table 6-41 -- Meaning of time_code

time_code	range of value	No. of bits	Mnemonic
time_code_hours	0 - 23	5	uimsbf
time_code_minutes	0 - 59	6	uimsbf
marker_bit	1	1	bslbf
time_code_seconds	0 - 59	6	uimsbf

skip_frames: This is a 1-bit flag which when set to '1' indicates that information follows this bit field that indicates the number of skipped frames. When set to '0' no such information follows this bit field.

6.3.10.4 Decode frame rate and frame skip

frame_rate: This is an 8 bit unsigned integer indicating the reference frame rate of the sequence.

seconds: This is a 4 bit unsigned integer indicating the fractional reference frame rate. The frame rate is computed as follows frame rate = (frame_rate + seconds/16).

frequency_offset: This is a 1-bit flag which when set to '1' indicates that the frame rate uses the NTSC frequency offset of 1000/1001. This bit would typically be set when frame_rate = 24, 30 or 60, in which case the resulting frame rate would be 23.97, 29.94 or 59.97 respectively. When set to '0' no frequency offset is present. I.e. if (frequency_offset ==1) frame rate = (1000/1001) * (frame_rate + seconds/16).

number_of_frames_to_skip: This is a 4-bit unsigned integer indicating the number of frames skipped. If the number_of_frames_to skip is equal to 15 (pattern "1111") then another 4-bit word follows allowing to skip up to 29 frames(pattern " 11111110"). If the 8-bits pattern equals "11111111", then another 4-bits word will follow and so on, and the number of frames skipped is incremented by 30. Each 4-bit pattern of ' 1111' increments the total number of frames to skip with 15.

6.3.10.5 Decode new minmax

i_new_max[j]: This is a 5-bit unsigned integer used to scale the maximum value of the arithmetic decoder used in the I frame.

i_new_min[j]: This is a 5-bit unsigned integer used to scale the minimum value of the arithmetic decoder used in the I frame.

p_new_max[j]: This is a 5-bit unsigned integer used to scale the maximum value of the arithmetic decoder used in the P frame.

p_new_min[j]: This is a 5-bit unsigned integer used to scale the minimum value of the arithmetic decoder used in the P frame.

6.3.10.6 Decode viseme and expression

viseme_def: This is a 1-bit flag which when set to '1' indicates that the mouth FAPs sent with the viseme FAP may be stored in the decoder to help with FAP interpolation in the future.

init_face: This is a 1-bit flag which when set to '1' indicates that the neutral face may be modified within the neutral face constraints.

expression_def: This is a 1-bit flag which when set to '1' indicates that the FAPs sent with the expression FAP may be stored in the decoder to help with FAP interpolation in the future.

6.3.10.7 Decode viseme_segment and expression_segment

viseme_segment_select1q[k]: This is the quantized value of viseme_select1 at frame k of a viseme FAP segment.

viseme_segment_select2q[k]: This is the quantized value of viseme_select2 at frame k of a viseme FAP segment.

viseme_segment_blendq[k]: This is the quantized value of viseme_blend at frame k of a viseme FAP segment.

viseme_segment_def[k]: This is a 1-bit flag which when set to '1' indicates that the mouth FAPs sent with the viseme FAP at frame k of a viseme FAP segment may be stored in the decoder to help with FAP interpolation in the future.

viseme_segment_select1q_diff[k]: This is the prediction error of viseme_select1 at frame k of a viseme FAP segment.

viseme_segment_select2q_diff[k]: This is the prediction error of viseme_select2 at frame k of a viseme FAP segment.

viseme_segment_blendq_diff[k]: This is the prediction error of viseme_blend at frame k of a viseme FAP segment.

expression_segment_select1q[k]: This is the quantized value of expression_select1 at frame k of an expression FAP segment.

expression_segment_select2q[k]: This is the quantized value of expression_select2 at frame k of an expression FAP segment.

expression_segment_intensity1q[k] This is the quantized value of expression_intensity1 at frame k of an expression FAP segment

expression_segment_intensity2q[k] This is the quantized value of expression_intensity2 at frame k of an expression FAP segment

expression_segment_select1q_diff[k]: This is the prediction error of expression_select1 at frame k of an expression FAP segment.

expression_segment_select2q_diff[k]: This is the prediction error of expression_select2 at frame k of an expression FAP segment.

expression_segment_intensity1q_diff[k] This is the prediction error of expression_intensity1 at frame k of an expression FAP segment.

expression_segment_intensity2q_diff[k] This is the prediction error of expression_intensity2 at frame k of an expression FAP segment.

expression_segment_init_face[k] This is a 1-bit flag which indicates the value of init_face at frame k of an expression FAP segment.

expression_segment_def[k] This is a 1-bit flag which when set to '1' indicates that the FAPs sent with the expression FAP at frame k of a viseme FAP segment may be stored in the decoder to help with FAP interpolation in the future.

6.3.10.8 Decode i_dc, p_dc, and ac

dc_q: This is the quantized DC component of the DCT coefficients. For an intra FAP segment, this component is coded as a signed integer of either 16 bits or 31 bits. The DCT quantisation parameters of the 68 FAPs are specified in the following list:

```
DCTQP[1 - 68] = {1,    1,    7.5,   7.5,   7.5,   7.5,   7.5,   7.5,   7.5,   7.5,
                 7.5,  7.5,  7.5,   15,    15,    15,    15,    5,     10,    10,
                 10,   10,   425,   425,   425,   425,   5,     5,     5,     5,
                 7.5,  7.5,  7.5,   7.5,   7.5,   7.5,   7.5,   7.5,   20,    20,
                 20,   20,   10,    10,    10,    10,    255,   170,   255,   255,
                 7.5,  7.5,  7.5,   7.5,   7.5,   7.5,   7.5,   7.5,   7.5,   7.5,
                 15,   15,   15,    15,    10,    10,    10,    10}
```

For DC coefficients, the quantisation stepsize is obtained as follows:

qstep[i] = fap_scale[fap_quant_inex] * DCTQP[i] ÷ 3.0

dc_q_diff: This is the quantized prediction error of a DC coefficient of an inter FAP segment. Its value is computed by subtracting the decoded DC coefficient of the previous FAP segment from the DC coefficient of the current FAP segment. It is coded by a variable length code if its value is within [-255, +255]. Outside this range, its value is coded by a signed integer of 16 or 32 bits.

count_of_runs: This is the run length of zeros preceding a non-zero AC coefficient.

ac_q[i][next]: This is a quantized AC coefficients of a segment of FAPi. For AC coefficients, the quantisation stepsize is three times larger than the DC quantisation stepsize and is obtained as follows:

qstep[i] = fap_scale[fap_quant_inex] * DCTQP[i]

BAPs are decoded using the same process. For BAPs, DCT quantization parameters (DCTQP[i]) have the same value as BAP predictive coding step sizesas defined in Annex C.

6.3.10.9 Decode bap min max

bap_i_new_max[i] – This is a 5-bit unsigned integer used to scale the maximum value of the arithmetic decoder used in the I frame.

bap_i_new_min[i] – This is a 5-bit unsigned integer used to scale the minimum value of the arithmetic decoder used in the I frame.

bap_p_new_max[i] – This is a 5-bit unsigned integer used to scale the maximum value of the arithmetic decoder used in the P frame.

bap_p_new_min[i] – This is a 5-bit unsigned integer used to scale the minimum value of the arithmetic decoder used in the P frame.

6.3.11 3D Mesh Object

6.3.11.1 3D_Mesh_Object

3D_MO_start_code: This is a unique 16-bit code that is used for synchronization purpose. The value of this code is always '0000 0000 0010 0000'.

6.3.11.2 3D_Mesh_Object_Layer

3D_MOL_start_code: This is a unique 16-bit code that is used for synchronization purposes. The value of this code is always '0000 0000 0011 0000'.

mol_id: This 8-bit unsigned integer specifies a unique id for the mesh object layer. Value 0 indicates a base layer, and value larger than 0 a refinement layer. The first 3D_Mesh_Object_Layer immediately after a 3D_Mesh_Object_Header must have mold_id=0, and subsequent 3DMesh_Object_Layer's within the same 3D_Mesh_Object must have mold_id>0.

ce_SNHC_n_vertices: This is the number of vertices in the current resolution of the 3D mesh. Used to support computational graceful degradation.

ce_SNHC_n_triangles: This is the number of triangles in the current resolution of the 3D mesh. Used to support computational graceful degradation.

ce_SNHC_n_edges: This is the number of edges in the current resolution of the 3D mesh. Used to support computational graceful degradation.

6.3.11.3 3DMesh_Object_Base_Layer

3D_MOBL_start_code: This is a code of length 16 that is used for synchronization purposes. It also indicates three different partition types for error resilience.

Table V2 - 5 -- Definition of partition type information

3D_MOBL_start_code	partition type	Meaning
'0000 0000 0011 0001'	partition_type_0	One or more groups of vg, tt and td.
'0000 0000 0011 0011'	partition_type_1	One or more vgs
'0000 0000 0011 0100'	partition_type_2	One pair of tt and td.

mobl_id: This 8-bit unsigned integer specifies a unique id for the mesh object component.

one_bit: This boolean value is always true. This value is used for byte alignment.

last_component: This boolean value indicates if there are more connected components to be decoded. If **last_component** is '1', then the last component has been decoded. Otherwise there are more components to be decoded. This field is arithmetic coded

codap_last_vg – This boolean value indicates if the current vg is the last one in the partition. The value is false if there are more vgs to be decoded in the partition.

codap_vg_id: This unsigned integer indicates the id of the vertex graph corresponding to the current simple polygon in partition_type_2. The length of this value is a log scaled value of the vg_number of vg decoded from the previous partition_type_1. If there is only one vg in the previous partition_type_1,

codap_left_bloop_idx: This unsigned integer indicates the left starting index, within the bounding loop table of a connected component, for the triangles that are to be reconstructed in a partition. The length of this value is the log scaled value of the size of the bounding loop table.

codap_right_bloop_idx: This unsigned integer indicates the right starting index, within the bounding loop table of a connected component, for the triangles that are to be reconstructed in a partition. The length of this value is the log scaled value of the size of the bounding loop table.

codap_bdry_pred: This boolean value denotes how to predict geometry and photometry information that are in common with two or more partitions. If **codap_bdry_pred** is '1', the restricted boundary prediction mode is used, otherwise, the extended boundary prediction mode is used.

6.3.11.4 3DMesh_Object_Header

ccw: This boolean value indicates if the vertex ordering of the decoded faces follows a counter clock-wise order.

convex: This boolean value indicates if the model is convex.

solid: This boolean value indicates if the model is solid.

creaseAngle: This 6-bit unsigned integer indicates the crease angle.

6.3.11.5 coord_header

coord_binding: This 2 bit unsigned integer indicates the binding of vertex coordinates to the 3D mesh. Table V2 - 6 shows the admissible values for **coord_binding**..

Table V2 - 6 -- Admissible values for coord_binding

coord_binding	binding
00	forbidden
01	bound_per_vertex
10	forbidden
11	forbidden

coord_bbox : This boolean value indicates whether a bounding box is provided for the geometry. If no bounding box is provided, a default bounding box is used. The default bounding box is defined as **coord_xmin**=0, **coord_ymin**=0, **coord_zmin**=0, and **coord_size**=1.

coord_xmin, **coord_ymin**, **coord_zmin**: These floating point values indicates the lower left corner of the bounding box in which the geometry lies.

coord_size: This floating point value indicates the size of the bounding box.

coord_quant: This 5-bit unsigned integer indicates the quantisation step used for geometry. The minimum value of coord_quant is 1 and the maximum is 24.

coord_pred_type: This 2-bit unsigned integer indicates the type of prediction used to reconstruct the vertex coordinates of the mesh. Table V2 - 7 shows the admissible values for coord_pred_type .

Table V2 - 7 -- Admissible values for coord_pred_type

coord_pred_type	prediction type
00	no_prediction
01	forbidden
10	parallelogram_prediction
11	reserved

coord_nlambda: This 2-bit unsigned integer indicates the number of ancestors used to predict geometry. The only admissible value of **coord_nlambda** is 3. Table V2 - 8 shows the admissible values as a function of **coord_pred_type.**

Table V2 - 8 -- Admissible values for coord_nlambda as a function of coord_prediction type

coord_pred_type	coord_nlambda
00	not coded
10	3

coord_lambda: This signed fixed-point number indicates the weight given to an ancestor for prediction. The number of bits used for this field is equal to **coord_quant** + 3. The 3 leading bits represent the integer part, and the **coord_quant** remaining bits the fractional part.

6.3.11.6 normal_header

normal_binding: This 2 bit unsigned integer indicates the binding of normals to the 3D mesh. The admissible values are described in Table V2 - 9.

Table V2 - 9 -- Admissible values for normal_binding

normal_binding	binding
00	not_bound
01	bound_per_vertex
10	bound_per_face
11	bound_per_corner

normal_bbox: This boolean value should always be false ('0').

normal_quant: This 5-bit unsigned integer indicates the quantisation step used for normals. The minimum value of normal_quant is 3 and the maximum is 31.

normal_pred_type: This 2-bit unsigned integer indicates how normal values are predicted. Table V2 - 10 shows the admissible values, and Table V2 - 11 shows admissible values as a function of normal_binding.

Table V2 - 10 -- Admissible values for normal_pred_type

normal_pred_type	prediction type
00	no_prediction
01	tree_prediction
10	parallelogram_prediction
11	reserved

Table V2 - 11 -- Admissible combinations of normal_binding and normal_pred_type

normal_binding	normal_pred_type
not_bound	not coded
bound_per_vertex	no_prediction, parallelogram_prediction
bound_per_face	no_prediction, tree_prediction
bound_per_corner	no_prediction, tree_prediction

normal_nlambda: This 2-bit unsigned integer indicates the number of ancestors used to predict normals. Admissible values of **normal_nlambda** are 1, 2, and 3. Table V2 - 12 shows admissible values as a function of **normal_pred_type.**

Table V2 - 12 -- Admissible values for normal_nlambda as a function of normal_prediction type

normal_pred_type	normal_nlambda
no_prediction	not coded
tree_prediction	1, 2, 3
parallelogram_prediction	3

normal_lambda: This signed fixed-point indicates the weight given to an ancestor for prediction. The number of bits used for normal_lambda is (normal_quant-3)/2+3. The 3 leading bits represent the integer part, and the **normal_quant** remaining bits the fractional part.

6.3.11.7 color_header

color_binding: This 2 bit unsigned integer indicates the binding of colors to the 3D mesh. Table V2 - 13 shows the admissible values.

Table V2 - 13 – Admissible values for color_binding

color_binding	Binding
00	not_bound
01	bound_per_vertex
10	bound_per_face
11	bound_per_corner

color_bbox: This boolean indicates if a bouding box for colors is given. If no bounding box is provided, a default bounding box is used. The default bounding box is defined as **color_rmin**=0, **color_gmin**=0, **color_bmin**=0, and **color_size**=1.

color_rmin, **color_gmin**, **color_bmin**: These floating point values give the position of the lower left corner of the bounding box in RGB space.

color_size: This floating point value gives the size of the color bounding box.

color_quant: This 5-bit unsigned integer indicates the quantisation step used for colors. The minimum value of color_quant is 1 and the maximum is 16.

color_pred_type: This 2-bit unsigned integer indicates how colors are predicted. Table V2 - 14 shows the admissible values, and Table V2 - 15 shows admissible values as a function of color_binding.

Table V2 - 14 – Admissible values for color_pred_type

color_pred_type	prediction type
00	no_prediction
01	tree_prediction
10	parallelogram_prediction
11	reserved

Table V2 - 15 – Admissible combinations of color_binding and color_pred_type

color_binding	color_pred_type
not_bound	not coded

bound_per_vertex	no_prediction, parallelogram_prediction
bound_per_face	no_prediction, tree_prediction
bound_per_corner	no_prediction, tree_prediction

color_nlambda: This 2-bit unsigned integer indicates the number of ancestors used to predict normals. Admissible values of **color_nlambda** are 1, 2, and 3. Table V2 - 16 shows admissible values as a function of **normal_pred_type.**

Table V2 - 16 -- Admissible values for color_nlambda as a function of color_prediction type

color_pred_type	color_nlambda
no_prediction	not coded
tree_prediction	1, 2, 3
parallelogram_prediction	3

color_lambda : This signed fixed-point indicates the weight given to an ancestor for prediction. The number of bits used for this field is equal to **color_quant** + 3. The 3 leading bits represent the integer part, and the **normal_quant** remaining bits the fractional part.

6.3.11.8 texCoord_header

texCoord_binding: This 2 bit unsigned integer indicates the binding of texture coordinates to the 3D mesh . Table V2 - 17 describes the admissible values.

Table V2 - 17 -- Admissible values for texCoord_binding

texCoord_binding	Binding
00	not_bound
01	bound_per_vertex
10	forbidden
11	bound_per_corner

texCoord_bbox: This boolean value indicates if a bounding box for texture coordinates is given. If no bounding box is provided, a default bounding box is used. The default bounding box is defined as **texCoord_umin**=0, **texCoord_vmin**=0, and **texCoord_size**=1.

texCoord_umin, **texCoord_vmin**: These floating point values give the position of the lower left corner of the bounding box in 2D space.

texCoord_size : This floating point value gives the size of the texture coordinate bounding box.

texCoord_quant: This 5-bit unsigned integer indicates the quantisation step used for texture coordinates. The minimum value of texCoord_quant is 1 and the maximum is 16.

texCoord_pred_type: This 2-bit unsigned integer indicates how colors are predicted. Table V2 - 18 shows the admissible values, and Table V2 - 19 shows admissible values as a function of texCoord_binding.

Table V2 - 18 -- Admissible values for texCoord_pred_type

texCoord_pred_type	prediction type
00	no_prediction
01	forbidden

10	parallelogram_prediction
11	reserved

Table V2 - 19 -- Admissible combinations of texCoord_binding and texCoord_pred_type

texCoord_binding	**texCoord_pred_type**
not_bound	not coded
bound_per_vertex	no_prediction, parallelogram_prediction
bound_per_corner	no_prediction, tree_prediction

texCoord_nlambda: This 2-bit unsigned integer indicates the number of ancestors used to predict normals. Admissible values of **texCoord_nlambda** are 1, 2, and 3. Table V2 - 20 shows admissible values as a function of **texCoord_pred_type.**

Table V2 - 20 -- Admissible values for texCoord_nlambda as a function of texCoord_prediction type

texCoord_pred_type	**texCoord_nlambda**
not_prediction	not coded
tree_prediction	1, 2, 3
parallelogram_prediction	3

texCoord_lambda: This signed fixed-point indicates the weight given to an ancestor for prediction. The number of bits used for this field is equal to **texCoord_quant** + 3. The 3 leading bits represent the integer part, and the **texCoord_quant** remaining bits the fractional part.

6.3.11.9 ce_SNHC_**header**

ce_SNHC_n_proj_surface_spheres: The number of Projected Surface Spheres. Typically, this number is equal to 1.

ce_SNHC_x_coord_center_point: The x-coordinate (in 32-bit IEEE floating point format) of the center point (typically the gravity point of the object) of the Projected Surface Sphere.

ce_SNHC_y_coord_center_point: The y-coordinate (in 32-bit IEEE floating point format) of the center point (typically the gravity point of the object) of the Projected Surface Sphere.

ce_SNHC_z_coord_center_point: The z-coordinate (in 32-bit IEEE floating point format) of the center point (typically the gravity point of the object) of the Projected Surface Sphere.

ce_SNHC_normalized_screen_distance_factor: This indicates where the virtual screen is placed, compared to the radius of the Projected Surface Sphere. The distance between the center point of the Projected Surface Sphere and the virtual screen is equal to *ce_SNHC_radius/(ce_SNHC_normalized_screen_distance_factor+1)*. Note that *ce_SNHC_radius* is specified for each Projected Surface Sphere, while *ce_SNHC_normalized_screen_distance_factor* is specified only once.

ce_SNHC_radius: The radius (in 32-bit IEEE floating point format) of the Projected Surface Sphere.

ce_SNHC_min_proj_surface: The minimal projected surface value (in 32-bit IEEE floating point format) on the corresponding Projected Surface Sphere. This value is often (but not necessarily) equal to one of the *ce_SNHC_proj_surface* values.

ce_SNHC_n_proj_points: The number of points on the Projected Surface Sphere in which the projected surface will be transmitted. For all other points, the projected surface is determined by linear interpolation. *ce_SNHC_n_proj_points* is typically small (e.g. 20) for the first Projected Surface Sphere and very small (e.g. 3) for additional Projected Surface Spheres.

ce_SNHC_sphere_point_coord: This indicates the index of the point position in a octahedron, as explained in "inverse quantisation" section (see subclause 7.13.8.4).

ce_SNHC_proj_surface: The projected surface (in 32-bit IEEE floating point format) in the point specified by *ce_SNHC_sphere_point_coord*.

6.3.11.10 connected component

has_stitches: This boolean value indicates if stitches are applied for the current connected component (within itself or between the current component and connected components previously decoded) This field is arithmetic coded.

6.3.11.11 vertex_graph

vg_simple: This boolean value indicates if the current vertex graph is simple. A simple vertex graph does not contain any loop. This field is arithmetic coded.

vg_last: This boolean value indicates if the current run is the last run starting from the current branching vertex. This field is not coded for the first run of each branching vertex, i.e. when the skip_last variable is true. When not coded the value of **vg_last** for the current vertex run is considered to be false. This field is arithmetic coded.

vg_forward_run: This boolean value indicates if the current run is a new run. If it is not a new run, it is a previously traversed run, indicating a loop in the graph. This field is arithmetic coded.

vg_loop_index: This unsigned integer indicates the index of run to which the current loop connects. Its unary representation (see Table V2 - 21) is arithmetic coded. If the variable openloops is equal to **vg_loop_index**, the trailing '1' in the unary representation is omitted.

Table V2 - 21 -- Unary representation of the vg_loop_index field

vg_loop_index	unary representation
0	1
1	01
2	001
3	0001
4	00001
5	000001
6 ... openloops-1	0000001 openloops-1 0's

vg_run_length: This unsigned integer indicates the length of the current vertex run. Its unary representation (see Table V2 - 22) is arithmetic coded.

Table V2 - 22 -- Unary representation of the vg_run_length field

vg_run_length	unary representation
1	1

2	01
3	001
4	0001
5	00001
6	000001
7	0000001
8	00000001
N	n-1 0's followed by 1

vg_leaf: This boolean value indicates if the last vertex of the current run is a leaf vertex. If it is not a leaf vertex, it is a branching vertex. This field is arithmetic coded.

vg_loop: This boolean value indicates if the leaf of the current run connects to a branching vertex of the graph, indicating a loop. This field is arithmetic coded.

6.3.11.12 stitches

stitch_cmd: This boolean value indicates if a stitching command of the type PUSH, POP or GET is associated to the current vertex. This field is arithmetic coded.

stitch_pop_or_get: This boolean value indicates if a stitching command of the type POP or GET is associated to the current vertex. This field is arithmetic coded.

stitch_pop: This boolean value indicates if a stitching command of the type POP is associated to the current vertex. This field is arithmetic coded.

stitch_stack_index: This unsigned integer value indicates the depth in the anchor stack where the anchor which the current vertex will be stitched to is located. This field is arithmetic coded.

stitch_incr_length: This integer value indicates the incremental length of the current stitch that must be added or subtracted to the length that is currently stored at the anchor. This field is arithmetic coded.

stitch_incr_length_sign: This boolean value indicates if the stitch_incr_length is negative. This field is arithmetic coded.

stitch_push: This boolean value indicates if the current vertex must be pushed into the stack of anchors. This field is arithmetic coded.

stitch_reverse: This boolean value indicates whether the current vertex must be stitched to its anchor using a reverse stitch as opposed to a forward stitch which is the default behavior. This field is arithmetic coded.

stitch_length: This unsigned integer value. This field is arithmetic coded.

6.3.11.13 triangle_tree

branch_position: This integer variable is used to store the last branching triangle in a partition.

tt_run_length: This unsigned integer indicates the length of the current triangle run. Its unary representation (see Table V2 - 23) is arithmetic coded.

Table V2 - 23 – Unary representation of the tt_run_length field

tt_run_length	unary representation
1	1
2	01
3	001
4	0001
5	00001
6	000001
7	0000001
8	00000001
N	n-1 0's followed by 1

tt_leaf: This boolean value indicates if the last triangle of the current run is a leaf triangle. If it is not a leaf triangle, it is a branching triangle. This field is arithmetic coded.

triangulated: This boolean value indicates if the current component contains triangles only. This field is arithmetic coded.

marching_triangle: This boolean value is determined by the position of the triangle in the triangle tree. If **marching_triangle** is 0, the triangle is a leaf or a branch. Otherwise, the triangle is a run.

marching_edge: This boolean value indicates the marching edge of an edge inside a triangle run. If **marching_edge** is false, it stands for a march to the left, otherwise it stands for a march to the right. This field is arithmetic coded.

polygon_edge: This boolean value indicates whether the base of the current triangle is an edge that should be kept when reconstructing the 3D mesh object. If the base of the current triangle is not kept, the edge is discarded. This field is arithmetic coded.

codap_branch_len: This unsigned integer indicates the length of the next branch to be traversed. The length of this value is the log scaled value of the size of the bounding loop table.

6.3.11.14 triangle

td_orientation: This boolean value informs the decoder the traversal order of tt/td pair at a branch. This field is arithmetic coded. Table V2 - 24 shows the admissible values.

Table V2 - 24 -- Admissible values for td_orientation

td_orientation	traversal order
0	right branch first
1	left branch first

visited: This variable indicates if the current vertex has been visited or not. When **codap_bdry_pred** is '1', visited is true for the vertices visited in the current partition. However, when **codap_bdry_pred** is '0', visited is true for the vertices visited in the previous partitions as well as in the current partition.

vertex_index: This variable indicates the index of the current vertex in the vertex array.

no_ancestors: This boolean value is true if there are no ancestors to use for prediction of the current vertex.

coord_bit: This boolean value indicates the value of a geometry bit. This field is arithmetic coded.

coord_leading_bit This boolean value indicates the value of aleading geometry bit. This field is arithmetic coded.

coord_sign_bit This boolean value indicates the sign of a geometry sample. This field is arithmetic coded.

coord_trailing_bit: This boolean value indicates the value of a trailing geometry bit. This field is arithmetic coded.

normal_bit: This boolean value indicates the value of a normal bit. This field is arithmetic coded.

normal_leading_bit: This boolean value indicates the value of a leading normal bit. This field is arithmetic coded.

normal_sign_bit: This boolean value indicates the sign of a normal sample. This field is arithmetic coded.

normal_trailing_bit: This boolean value indicates the value of a trailing normal bit. This field is arithmetic coded.

color_bit: This boolean value indicates the value of a color bit. This field is arithmetic coded.

color_leading_bit: This boolean value indicates the value of a leading color bit. This field is arithmetic coded.

color_sign_bit: This boolean value indicates the sign of a color sample. This field is arithmetic coded.

color_trailing_bit: This boolean value indicates the value of a trailing color bit. This field is arithmetic coded.

texCoord_bit: This boolean value indicates the value of a texture bit. This field is arithmetic coded.

texCoord_leading_bit: This boolean value indicates the value of a leading texture bit. This field is arithmetic coded.

texCoord_sign_bit: This boolean value indicates the sign of a texture sample. This field is arithmetic coded.

texCoord_trailing_bit: This boolean value indicates the value of a trailing texture bit. This field is arithmetic coded.

6.3.11.15 3DMeshObject_Refinement_Layer

3D_MORL_start_code: This is a unique 16-bit code that is used for synchronization purpose. The value of this code is always '0000 0000 0011 0010'.

morl_id: This 8-bit unsigned integer specifies a unique id for the forest split component.

connectivity_update: This 2-bit variable indicates whether the forest split operation results in a refinement of the connectivity of the mesh or not.

Table V2-25 -- Admissible values for connectivity_update

connectivity_update	meaning
00	not_updated
01	fs_update
10	reserved
11	reserved

pre_smoothing: This boolean value indicates whether the current forest split operation uses a pre-smoothing step to globally predict vertex positions.

post_smoothing: This boolean value indicates whether the current forest split operation uses a post-smoothing step to remove quantisation artifacts.

stuffing_bit: This boolean value is always true.

other_update: This boolean value indicates whether updates for vertex coordinates and properties associated with faces and corners not incident to any tree of the forest follow in the bitstream or not.

other_update: this boolean value indicates whether updates for vertex coordinates and properties associated with faces and corners not incident to any tree of the fore st follow in the bitstream or not.

6.3.11.15.1 pre_smoothing_parameters

pre_smoothing_n: This integer value indicates the number of iterations of the pre-smoothing filter.

pre_smoothing_lambda: This float value is the first parameter of the pre-smoothing filter.

pre_smoothing_mu: This float value is the second parameter of the pre-smoothing filter.

6.3.11.15.2 post_smoothing_parameters

post_smoothing_n: This integer value indicates the number of iterations of the pre-smoothing filter.

post_smoothing_lambda: This float value is the first parameter of the pre-smoothing filter.

post_smoothing_mu: This float value is the second parameter of the pre-smoothing filter.

6.3.11.15.3 fs_pre_update

pfs_forest_edge: This boolean value indicates if an edge should be added to the forest built so far.

6.3.11.15.4 smoothing_constraints

smooth_with_sharp_edges: This boolean value indicates if data is included in the bitstream to mark smoothing discontinuity edges or not. If **sharp_edges** ==0 no edge is marked as a smoothing discontinuity edge. If smoothing discontinuity edges are marked, then both the pre-smoothing and post-smoothing filters take them into account.

smooth_with_fixed_vertices: This boolean value indicates if data is included in the bitstream to mark vertices which do not move during the smoothing process. If **smooth_with_fixed_vertices**==0 all vertices are allowed to move. If fixed vertices are marked, then both the pre-smoothing and post-smoothing filters take them into account.

smooth_sharp_edge: This boolean value indicates if a corresponding edge is marked as a smoothing discontinuity edge.

smooth_fixed_vertex: This boolean value indicates if a corresponding vertex is marked as a fixed vertex or not.

6.3.12 Upstream message

6.3.12.1 upstream_message

upstream_message_type: This 3-bit value indicates the type of the upstream information as shown in Table V2 -26. The meaning of each upstream type is as follows:

video_newpred: This upstream message conveys the decoding status of the receiver (decoder) for the NEWPRED mode. The NEWPRED is the error resilience tool by selecting the reference picture of the inter-frame coding according to the error condition of the network. This upstream message shows whether the decoder decode the forward video data correctly or not. This message returns corresponding to the VOP or Video Packet of the forward video data. Which type of message is required for the encoder is indicated in requested_upstream_message_type in the VOL header of the downstream data. The decoder definitions of NEWPRED are described in subclause s 7.14 and E.1.6.

SNHC_QoS: This upstream message conveys some information that reveals to the encoder (server) the performances of the decoder w.r.t. decoding and rendering 3D objects with different parameter settings, i.e. with a varying number of triangles, different screen coverages of the rendered objects and different rendering modes. Using this information, the encoder can restrict the 3D content to the capabilities of the decoder, using for instance mesh simplification and/or rendering mode selection.

Table V2 - 26 -- Meaning of upstream_message_type

upstream_message_type	meaning
000	reserved
001	video_newpred
010	SNHC_QoS
011-111	reserved

6.3.12.2 upstream_video_newpred

newpred_upstream_message_type : This indicates whether the corresponding NP segment is correctly decoded or not. Which type of message is required for the encoder is indicated in requested_upstream_message_type in the VOL header of the downstreamdata. In the other case, this indicates requesting intra refresh.

00: NP_NACK. It indicates the erroneous decoding of the NP segment.
01: NP_ACK. It indicates the correct decoding of the NP segment.
10: Intra refresh command.
11: Reserved.

unreliable_flag This field presents only if newpred_upstream_message_type is 'NP_NACK'. The unreliable_flag is set to 1 when a reliable value of vop_id is not available at the decoder. (When the NP segment is erred, a reliable vop_id may not be available at the decoder. On the other hand, a reliable vop_id is available, when the decoder cannot decode due to the luck of the reference picture.)

0: reliable
1: unreliable

vop_id: When the newpred_upstream_message_type is 'NP_NACK' or 'NP_ACK', this indicates the ID of VOP which is incremented by 1 whenever a VOP is encoded. The vop_id is copied from the vop_id field of the NP segment header in the corresponding forward channel data when the reliable vop_id is available. Otherwise, it may happen in the case of NP_NACK that the vop_id is incremented by 1 from the reliable vop_id of the previously received NP segment in the same location of the current NP segment.

When the newpred_upstream_message_type is 'Intra refresh command', this indicates the ID of Intra refresh which is incremented by 1 whenever new refresh is required. The length of this field is 4 bits in the Intra refresh case. In the case that Intra refresh command is repeatedly returned for the same error until the proper action corresponding to the previous Intra refresh command reaches, this ID is set to the same number as the previous Intra refresh command.

macroblock_number: The macroblock_number is the macroblock address of the start of the corresponding NP segment or the refresh area.

end_macroblock_number: This field is present only when the newpred_upstream_message_type is 'Intra refresh command'. The end_macroblock_number is the macroblock address of the end of the refresh area.

requested_vop_id_for_prediction: This field is present only if newpred_upstream_message_type is 'NP_NACK'. This indicates the requested vop_id of the NP segment for reference by the decoder. Typically it is the vop_id of the last correctly decoded NP segment in the same location of the current NP segment.

6.3.12.3 upstream_SNHC_QoS

screen_width: Screen width used during the calibration process. screen_width is expressed in number of pixels.

screen_height: Screen height used during the calibration process. screen_height is expressed in number of pixels.

n_rendering_modes: n_rendering_modes is the number of rendering modes for which information is transmitted.

rendering_mode_type: rendering_mode_type is the kind of rendering used during the calibration process for a particular performance curve. The different rendering types are coded according to the following table.

Table V2-27 -- Meaning of rendering_mode_type

rendering_mode_type	Rendering mode
0000	Wire-framed
0001	Flat shading
0010	Smooth shading
0011	Texture rendering
0100-1111	Reserved for later use

n_curves: n_curves is the number of performance curves transmitted for one particular rendering mode.

triangle_parameter: triangle_parameter is the number of triangles in units of 64 triangles.

n_points_on_curve: n_points_on_curve is the number of points specified for one particular performance curve.

screen_coverage_parameter: screen_coverage_parameter is the number of pixels, expressed in percentage of the screen size. 0x00 corresponds to 0%, while 0xFF corresponds to 100%. All other points are determined by linear interpolation.

frame_rate_value: frame_rate_value is the frame rate for one particular point on one particular curve. For achieving enough precision, 12 bits are used. The 8 Most Significant Bits represent the integer value, the 4 Least Significant Bits represent the fractional value.

7 The visual decoding process

This clause specifies the decoding process that the decoder shall perform to recover visual data from the coded bitstream. As shown in Figure 7-1, the visual decoding process includes several decoding processes such as shape-motion-texture decoding, still texture decoding, mesh decoding, and face decoding processes. After decoding the coded bitstream, it is then sent to the compositor to integrate various visual objects.