ETSI TS 103 491 V1.2.1 (2019-05)

DTS-UHD Audio Format;
Delivery of Channels, Objects and Ambisonic Sound Fields

EBU

Reference
RTS/JTC-DTS-UHDv121

Keywords
audio, codec, object audio

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

The present document can be downloaded from:
http://www.etsi.org/standards-search

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the prevailing version of an ETSI deliverable is the one made publicly available in PDF format at www.etsi.org/deliver.

Users of the present document should be aware that the document may be subject to revision or change of status.
Information on the current status of this and other ETSI documents is available at
https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx

If you find errors in the present document, please send your comment to one of the following services:
https://portal.etsi.org/People/CommiteeSupportStaff.aspx

Copyright Notification

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.
The content of the PDF version shall not be modified without the written authorization of ETSI.
The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2019.
© European Broadcasting Union 2019.
All rights reserved.

DECT™, **PLUGTESTS**™, **UMTS**™ and the ETSI logo are trademarks of ETSI registered for the benefit of its Members.
3GPP™ and **LTE**™ are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners.
oneM2M™ logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners.
GSM® and the GSM logo are trademarks registered and owned by the GSM Association.

Contents

Intellectual Property Rights 23
Foreword 23
Modal verbs terminology 23
1 Scope 24
2 References 24
2.1 Normative references 24
2.2 Informative references 25
3 Definition of terms, symbols, abbreviations and document conventions 25
3.1 Terms 25
3.2 Symbols 26
3.3 Abbreviations 26
3.4 Document Conventions 27
4 DTS-UHD System Overview 27
4.1 Overview 27
4.2 Stream Construction 28
4.2.1 Construction of a DTS-UHD Audio Frame 28
4.2.2 Frame Table of Contents (FTOC) 28
4.2.3 Sync Frames 29
4.2.4 Non-sync Frames (Predictive Frames) 29
4.3 Carriage of Metadata 29
4.3.1 Organization of Metadata 29
4.3.2 Metadata Chunks 30
4.3.3 Fundamental Components of the Metadata Chunk 31
4.3.3.1 Metadata Chunk: Data 31
4.3.3.2 Reserved and Byte Align Fields 31
4.3.3.3 Metadata Chunk CRC Word 31
4.4 Audio Chunks 31
4.5 Organization of Streams 33
4.5.1 Objects, Object Groups, Presentations 33
4.5.2 Properties of Objects 33
4.5.3 Properties of Object Groups 33
4.5.4 Audio Presentations 34
5 DTS-UHD Header Tables and Helper Functions 37
5.1 Overview 37
5.2 Constants, Tables and Helper Functions 37
5.2.1 Fixed Point Constants 37
5.2.2 Look-up Tables 37
5.2.2.1 Scale Factor Table 37
5.2.2.2 Long Term Loudness Measure Table 37
5.2.2.3 Per-Object Long Term Loudness Measure Table 37
5.2.2.4 Quantization Table for DRC Fast Attack Smoothing Constant 38
5.2.2.5 Quantization Table for DRC Fast Release Smoothing Constant 38
5.2.2.6 Quantization Table for DRC Slow to Fast Threshold 38
5.2.2.7 Inverse Quantization Table for the Exponential Window Smoothing Parameter Lambda 38
5.2.3 Helper Functions 39
5.2.3.1 ExtractVarLenBitFields 39
5.2.3.2 UpdateCode 39
5.2.3.3 CountBitsSet_to_1 41
5.2.3.4 GetPtrToCurrentByte 41
5.2.3.5 GetBitCounter 41
5.2.3.6 AdvanceBitReadPointer 41
5.3 Interfaces for Extracting Metadata Frames 42
5.3.1 Overview of APIs 42
5.3.2 Audio Presentation Selection 42

5.3.3 Play Default Presentation....42
5.3.4 Play Selected Presentation....43
5.3.5 Play from a List of Objects....43
6 DTS-UHD Metadata Header Parsing....43
6.1 Overview....43
6.2 UnpackMDFrame....44
6.3 Functions and Tables Supporting UnpackMDFrame....46
6.3.1 ResetMDChunks....46
6.3.2 Metadata Chunk Types....46
6.3.3 Audio Presentation Index....47
6.3.4 ExtractMDChunkObjIDList....47
6.3.5 Parameters in ExtractMDChunkObjIDList....48
6.3.5.1 m_unNumObjects....48
6.3.5.2 unNumBitsforObjID....48
6.3.5.3 m_unObjectID....48
6.3.6 Update Object List....48
6.3.7 Reserved and Byte Alignment Fields....49
6.3.8 Metadata Chunk CRC....50
6.4 Frame Table of Contents (FTOC)....51
6.4.1 Overview of the FTOC....51
6.4.2 FTOC Class Parameters....51
6.4.3 Extract FTOC....52
6.4.4 Parameters for FTOC....52
6.4.4.1 unSyncWord....52
6.4.4.2 bSyncFrameFlag....53
6.4.4.3 unFTOCPayloadinBytes....53
6.4.5 Stream Parameters....53
6.4.6 Parameters for ExtractStreamParams....54
6.4.6.1 Full Channel-based Mix Flag....54
6.4.6.2 CRC Test....54
6.4.6.3 Stream Version Number....54
6.4.6.4 Base Duration....55
6.4.6.5 Frame Duration....55
6.4.6.6 Clock Rate....55
6.4.6.7 Time Code Parameters....55
6.4.6.8 Audio Sampling Rate....56
6.4.6.9 Number of Audio Samples....56
6.4.6.10 Reserved Flag....56
6.4.6.11 Object Interactivity Limits Present Flag....56
6.4.7 ResolveAudPresParams....56
6.4.8 Parameter for ResolveAudPresParams....57
6.4.8.1 Number of Audio Presentations....57
6.4.9 Extract Audio Presentation Parameters....57
6.4.10 Parameters for ExtractAudPresParams....58
6.4.10.1 bAudPresSelectableFlag....58
6.4.10.2 unDepAuPresMask....59
6.4.10.3 Explicit Object List Present Flag....59
6.4.11 ExtractExplicitObjectsLists....59
6.4.12 Parameters for ExtractExplicitObjectLists....60
6.4.12.1 Explicit Object List Update Flag....60
6.4.12.2 Explicit Object List Mask....60
6.4.12.3 Metadata Describing Presentation Scaling for Different Playback Configurations....60
6.4.12.4 Extract and Save All Audio Presentation Parameters....60
6.4.13 Navigation Parameters for Metadata and Audio Chunks....61
6.4.14 Parameters for ExtractChunkNaviData....62
6.4.14.1 Number of Metadata Chunks....62
6.4.14.2 Sizes of Metadata Chunks....62
6.4.14.3 Metadata Chunks CRC Flag....63
6.4.14.4 Number of Audio Chunks....63
6.4.14.5 Audio Chunk Handler ResetList....63
6.4.14.6 ResetAudioChunkPresentFlags....63

6.4.14.7 Audio Chunk Index 63
6.4.14.8 FindAudioChunkListIndex 63
6.4.14.9 Audio Chunk ID Present Flag 64
6.4.14.10 Audio Chunk ID 64
6.4.14.11 Audio Chunk Size in Bytes 64
6.4.14.12 Purge List 64
6.4.15 Peak Bit Rate Smoothing Parameters 65
6.4.16 Parameters for ExtractPBRSmoothParams 66
6.4.16.1 m_bVBRSmoothingBufferEnabled 66
6.4.16.2 nuBitsforCumACPayload 66
6.4.16.3 m_nuAudioChunksCumulSmoothPayload 66
6.4.16.4 m_bACFirstXLLSyncPresent 66
6.4.16.5 nuVBRSmoothingBuffSzkB 66
6.4.16.6 m_nuInitACXLLDecDlyFrames 66
6.4.16.7 m_nuACFirstXLLSyncOffsInDWORDS 67
6.4.17 Reserved Fields 67
6.4.18 FTOC CRC Word 67

7 Metadata Chunk (ID = 0x01) 68
7.1 Overview 68
7.2 Object Class Description and Handler Functions 68
7.3 Functions Called from UnpackMDFrame 69
7.3.1 ApplyExplObjList2ObjActMask 69
7.3.2 ApplyGroupDefaultObjectActivityMask 70
7.3.3 Find Object Index 71
7.4 Extract_Main_0x01_MDChunk 71
7.5 Functions Called from Extract_Main_0x01_MDChunk 72
7.5.1 Presentation Scaling Parameters 72
7.5.2 Parameters for ExtractPresScalingParams 73
7.5.2.1 bOutScalePresent 73
7.5.2.2 Presentation Scale 73
7.5.3 m_bMFDistrStaticMDPresent 73
7.6 Static Metadata Distributed Over Multiple Frames 74
7.7 Parameters for ExtractMultiFrameDistribStaticMD 75
7.7.1 m_unNumStaticMDPackets 75
7.7.2 m_unStaticMDPacketByteSize 75
7.7.3 m_bStaticMetadataUpdtFlag 76
7.7.4 Metadata Packet Payload 76
7.7.5 Static Loudness and Dynamics Metadata 76
7.7.6 Parameters for ExtractStaticLandDParams 78
7.7.6.1 m_bNominalLD_DescriptionFlag 78
7.7.6.2 Integrated Loudness Parameters 78
7.7.6.3 m_ucNumLongTermLoudnessMsrmSets 78
7.7.6.4 Long Term Loudness Measure Parameter Set 78
7.7.6.5 Parameters for ExtractLTLMParamSet 79
7.7.6.5.1 m_rLoudness 79
7.7.6.5.2 m_ucAssociatedAssetType 79
7.7.6.5.3 m_ucLoudnessMsrmType 80
7.7.6.6 m_bIsLTLoudnMsrsmOffLine 80
7.7.6.7 m_bCustomDRCCurveMDPresent 81
7.7.6.8 Custom DRC Curves 81
7.7.6.9 Parameters for ExtractCustomDRCCurves 82
7.7.6.9.1 m_ucDRCCurveIndex 82
7.7.6.9.2 unDRCCurveCode 82
7.7.6.9.3 Get DRC Curve 83
7.7.6.10 m_bCustomDRCSmoothMDPresent 84
7.7.6.11 m_rFastAttack 84
7.7.6.12 m_rSlowAttack 84
7.7.6.13 m_rFastRelease 84
7.7.6.14 m_rSlowRelease 84
7.7.6.15 m_rAttackThreshld 84
7.7.6.16 m_rReleaseThreshld 85

7.7.6.17 Reserved and Byte Alignment Fields 85
7.8 Extraction of Object Metadata 85
7.8.1 Overview of Object Metadata 85
7.8.2 ExtractMetadataForObjects 85
7.8.3 Parameters for ExtractMetadataForObjects 86
7.8.3.1 m_bMixStudioParamsPresent 86
7.8.3.2 m_ucRadiusRefernceUnitSphereInMeters 87
7.8.3.3 m_unRefScreenHorizontalViewingAngleInDeg 87
7.8.3.4 m_ucRefScreenAspectRatio 87
7.8.4 Association of 3D Object Metadata to 3D Renderer Type 87
7.8.5 Parameters for CheckIfMDIsSuitableforImplObjRenderer 88
7.8.5.1 bMDUsedByAllRenderersFlag 88
7.8.5.2 ucRequiredRendererType 88
7.8.5.3 unNumBits2Skip 88
7.8.6 m_bObjStaticFlag 89
7.8.7 Extraction of 3D Object/Object Group Metadata 89
7.8.8 Object Group Definition 90
7.8.9 Parameters for ExtractGroupDefinition 91
7.8.9.1 m_bObjGrpActiveFlag 91
7.8.9.2 bGrpObjRefUpdFlag 91
7.8.9.3 m_ucNumObjReferencedInGroup 91
7.8.9.4 m_punObjGrpRefIDs[] 92
7.8.9.5 bUpdFlag 92
7.8.9.6 m_unWithinGrpObjActMask 92
7.8.9.7 m_bObjGroupMDExtensionPresent 92
7.8.9.8 unNumBitsToExtract 92
7.8.9.9 Object Group Metadata Extension Fields 92
7.8.10 3D Object Metadata 92
7.8.11 Parameters for ExtractObjectMetadata 95
7.8.11.1 m_bObjActiveFlag 95
7.8.11.2 m_ucObjRepresTypeIndex 95
7.8.11.3 m_ucObjectImportanceLevel 96
7.8.11.4 bObjTypeDescrPresent 96
7.8.11.5 m_unObjTypeDescrIndex 96
7.8.11.6 ucObjAudioChunkIndex 96
7.8.11.7 m_ucObjNaviWitinACIndex 97
7.8.11.8 m_bPerObjLTLoudnessMDPresent 97
7.8.11.9 Per-Object Loudness Metadata 97
7.8.11.10 Parameters for ExtractObjectMetadata 97
7.8.11.10.1 m_rObjIntegrLoudnessMsr 97
7.8.11.10.2 m_bMatchLDofReplacedObj 97
7.8.11.10.3 m_bMatchLDofSimilarObj 97
7.8.11.11 Object Interactivity Related Metadata 98
7.8.11.12 Parameters for ExtractObjectInteractMD 99
7.8.11.12.1 m_bObjInteractiveFlag 99
7.8.11.12.2 m_bObjInterLimitsFlag 99
7.8.11.12.3 m_unMaxInterObjGainBoostdB 99
7.8.11.12.4 m_unMaxInterObjGainAttendB 99
7.8.11.12.5 m_unObjInterPosMaxDeltaAzim 99
7.8.11.12.6 m_unObjInterPosMaxDeltaElev 100
7.8.11.13 Metadata for Channel Mask Parameters 100
7.8.11.14 Parameters for ExtractChMaskParams 100
7.8.11.14.1 m_ucChLayoutIndex 100
7.8.11.14.2 m_uint32ChActivityMask 101
7.8.11.14.3 Count the Number of Active Channels 102
7.8.11.15 Per Object Renderer Configuration Parameters 102
7.8.11.16 Parameters for ExtractRendererConfigParams 103
7.8.11.16.1 m_bEnhancedREConfigParamsPresent 103
7.8.11.16.2 m_bUseDivergApperObjSpread 103
7.8.11.16.3 m_bEnableSnaptoSpkrs 103
7.8.11.16.4 m_unSnap2SpkrSpherCapAngle 103
7.8.11.16.5 m_unRendrerExcludedSpkrsChMask 104

7.8.11.16.6 m_bAudioObjLinked2VisualObj 104
7.8.11.17 m_ucNumWaveFormsInObj 104
7.8.11.18 Metadata Describing Ambisonic Representation 104
7.8.11.19 Parameters for ExtractAmbisonicsMD 109
7.8.11.19.1 m_ucAmbixAdaptorMtrxIndicator 109
7.8.11.19.2 m_ucAmbisonicRepresOrder 110
7.8.11.19.3 nuWfToACNContribMask 110
7.8.11.19.4 Non-zero Entries of the Adaptor Matrix 110
7.8.11.20 m_bObjectRendExceptPresent 111
7.8.11.21 Object Properties with Multiple Updates Per Frame 111
7.8.11.22 Parameters for ExtractInvQuantMultiUpdtObjMD 113
7.8.11.22.1 Initialize Multiple Update Object Metadata 113
7.8.11.22.2 m_bPerSamplPeriodObjMDUpdFlag 114
7.8.11.22.3 m_ppObjGain 114
7.8.11.22.4 m_ucNum3DSourcesInObj 114
7.8.11.22.5 unsrc_index (3D Source Index) 114
7.8.11.23 m_rPerObjExpWinLambda 114
7.8.11.24 m_bObjecMDExtensionPresent 115
7.8.11.25 Size of Object Metadata Extension Fields 115
7.8.11.26 Object Metadata Extension Fields 115
7.8.11.27 Extraction of 3D Source Properties 115
7.8.11.28 Parameters for Extract3DSourceProperties 116
7.8.11.28.1 m_3DSrcTypeIndex 116
7.8.11.28.2 m_3DSrcRadius 117
7.8.11.28.3 m_3DSrcAzimuth 117
7.8.11.28.4 m_3DSrcElevation 118
7.8.11.28.5 m_3DSrcRadiusAux 118
7.8.11.28.6 m_3DSrcSpreadParamOne 118
7.8.11.28.7 m_3DSrcSpreadParamTwo 119
7.8.11.29 Extract Rendering Exception Object Metadata 119
7.8.11.30 Parameters for ExtractREObjectMD 121
7.8.11.30.1 bObjectRendExceptUpdateFlag 121
7.8.11.30.2 m_bUseREMForObjWithSameFullBWChMask 121
7.8.11.30.3 ucNumRESets 121
7.8.11.30.4 bContribCoeffsSigned 121
7.8.11.30.5 unREChannelMask 121
7.8.11.30.6 bExtendREtoSuperSetLayoutsFlag 122
7.8.11.30.7 Checking Applicability of the RE Parameter Set 122
7.8.11.30.8 nuREMixBusChContribMask 122
7.8.11.30.9 m_nRECoeffsMatrix[] 122
7.8.11.30.10 Metadata Packet Payload 123

8 ACE for DTS-UHD 123
8.1 Overview 123
8.2 Fundamental Bitstream Operations for ACE 124
8.2.1 Align 124
8.2.2 Skip 124
8.2.3 BitsUsed 125
8.2.4 BitsLeft 125
8.2.5 BitsTotal 125
8.2.6 ReadBit 125
8.2.7 ReadBool 126
8.2.8 ReadUint 126
8.2.9 ReadInt 126
8.2.10 ReadUnary 127
8.2.11 ReadUniform 127
8.2.12 ReadGolomb 127
8.2.13 ReadGolombLimited 128
8.2.14 ReadGolombWithParams 128
8.2.15 ReadVLC 129
8.2.16 ReadLimitsVLC 129
8.2.17 ReadNonUniformFiveTen 130

8.3 Auxiliary Functions 130
8.3.1 NumBits 130
8.3.2 Modular 131
8.3.3 NegativeRiceMap 131
8.3.4 PositiveRiceMap 131
8.3.5 CenterMap 132
8.3.6 Rand 132
8.4 ACE Data Types 133
8.4.1 FixedPoint 133
8.4.2 Fixed Point Instances 138

9 ACE Decoder 138
9.1 Overview 138
9.2 Bitstream 138
9.2.1 Bitstream Data and Method 138
9.2.2 read_bitstream 138
9.3 ACE Frames 139
9.3.1 Frame Data and Methods 139
9.3.2 read_decode_frame 139
9.3.3 Function Supporting read_decode_frame 140
9.3.3.1 read_padding 140
9.4 ACE Frame Header 140
9.4.1 Frame Header Data and Methods 140
9.4.2 read_frame_header 141
9.4.3 Functions used by read_frame_header 142
9.4.3.1 read_is_sync 142
9.4.3.2 read_enable_deemphasis 142
9.4.3.3 read_frame_duration 142
9.4.3.4 read_sampling_rate 143
9.4.3.5 read_num_stream_sets 143
9.4.3.6 read_has_padding 143
9.5 ACE StreamSetHeader 144
9.5.1 StreamSetHeader Data and Methods 144
9.5.2 read_streamset_header 145
9.5.3 Functions called from read_streamset_header 146
9.5.3.1 read_streamset_id 146
9.5.3.2 read_stream_information 146
9.5.3.3 read_lfe_stream_payload_size 147
9.5.3.4 read_mono_stream_payload_size 147
9.5.3.5 read_stereo_stream_payload_size 148
9.5.3.6 read_stream_bandwidth_mode 148
9.6 ACE Stream Set 148
9.6.1 Stream Set Data and Methods 148
9.6.2 read_decode_streamset 149
9.7 ACE LFEStreams 150
9.7.1 LFE Stream Data and Methods 150
9.7.2 read_decode_lfe_stream 150
9.7.3 LFEChannel 151
9.7.3.1 LFEChannel Data and Methods 151
9.7.3.2 read_decode_lfe_channel 152
9.7.3.3 Functions Called by read_lfe_channel 153
9.7.3.3.1 read_resolution 153
9.7.3.3.2 read_savings 153
9.7.3.3.3 read_dbnorm 154
9.7.3.3.4 read_predictor 154
9.7.3.3.5 read_lfe_decimated_buffer 154
9.8 ACE Streams 155
9.8.1 Stream Data and Methods 155
9.8.2 read_decode_mono_stream 157
9.8.3 read_decode_stereo_stream 158
9.8.4 read_decode_stream 158
9.9 Stream Preparation 160

9.9.1 read_coding_mode 160
9.9.2 compute_effective_channels 161
9.9.3 read_long_term_synthesis 161
9.9.4 read_vq_resolution_mode 162
9.9.5 read_mdct_mode 162
9.9.6 read_shf_enabled 163
9.9.7 read_thf_enabled 163
9.9.8 read_lognorm_primary 163
9.9.9 read_num_band_blocks 164
9.9.10 Functions Supporting read_num_band_blocks 165
9.9.10.1 map_tf_change 165
9.9.10.2 undo_differential 165
9.9.10.3 enforce_range 166
9.9.11 read_conditioning_level 166
9.9.12 read_allocation 166
9.9.13 Functions use by read_allocation 168
9.9.13.1 read_num_coded_bands 168
9.9.13.2 read_num_bands 169
9.9.13.3 read_allocation_model 169
9.9.13.4 read_alloc_model_parameter 169
9.9.13.5 read_alloc_model_delta 170
9.9.13.6 num_bits_step 170
9.9.13.7 read_num_ms_mono_bands 171
9.9.13.8 read_stereo_flags 171
9.9.13.9 do_allocation_model 173
9.9.13.10 do_allocation_model_step 173
9.9.13.11 compute_adjustments_from_lognorms 174
9.9.13.12 estimate_cost_from_adjustments 175
9.9.13.13 read_alloc_diff_steps 175
9.9.13.14 Functions Supporting read_alloc_diff_steps 176
9.9.13.14.1 remove_guaranteed_bits 176
9.9.13.14.2 compute_delta_alphabets 177
9.9.13.14.3 restore_guaranteed_bits 177
9.9.13.15 compute_alloc_from_foundation 177
9.9.13.16 compute_first_lognorm_refinement_alloc 178
9.9.13.17 Function Used by compute_first_lognorm_refinement_alloc 179
9.9.13.17.1 adjust_refinement 179
9.9.14 do_first_lognorm_refinement 180
9.9.15 Functions Supporting do_first_lognorm_refinement 180
9.9.15.1 do_lognorm_refinement 180
9.9.15.2 compute_refinement_correction 181
9.9.15.3 transfer_lognorms 181
9.9.16 do_late_lognorm_refinement 182
9.9.17 Function Supporting do_late_lognorm_refinement 182
9.9.17.1 compute_late_lognorm_refinement_alloc 182
9.9.18 apply_lognorms 183
9.10 Decoding Algorithms 184
9.10.1 Band Reconstruction 184
9.10.2 Functions to Perform Band Reconstruction 184
9.10.2.1 Band Reconstruction Constants and Definitions 184
9.10.2.2 read_all_bands 184
9.10.2.3 Functions Supporting read_all_bands 186
9.10.2.3.1 distribute_running_balance 186
9.10.2.3.2 read_one_band_mono 186
9.10.2.3.3 save_for_late_refinement 187
9.10.3 Stereo 187
9.10.3.1 Stereo Overview 187
9.10.3.2 Stereo Constants and Definitions 187
9.10.3.3 read_stereo_coding_type 188
9.10.3.4 read_one_band_leftright 188
9.10.3.5 Functions Called from read_one_band_left_right 189
9.10.3.5.1 compute_leftright_allocations 189

9.10.3.5.2 compute_region 190
9.10.3.6 read_one_band_midside 190
9.10.3.7 midside_to_leftright 191
9.10.4 Vector Quantization (VQ) 192
9.10.4.1 Overview of VQ 192
9.10.4.2 VQ Constants and Definitions 192
9.10.4.3 read_split_vector 192
9.10.4.4 Functions Supporting read_split_vector 194
9.10.4.4.1 is_split_required 194
9.10.4.4.2 compute_split_parameters 194
9.10.4.4.3 compute_quant_level 195
9.10.4.5 read_nonsplit_vector 196
9.10.4.6 make_unsigned_pyramid_vector 197
9.10.4.7 Reconditioning 197
9.10.4.7.1 recondition 197
9.10.4.7.2 make_matrix_4 198
9.10.4.7.3 apply_matrix 199
9.10.4.7.4 flip_vector 199
9.10.4.7.5 interleave_vector 200
9.10.4.8 Spectral Hole Fill 200
9.10.4.8.1 Overview of Spectral Hole Fill 200
9.10.4.8.2 do_spectral_hole_fill (informative) 200
9.10.4.9 Auxiliary functions 201
9.10.4.9.1 factorial 201
9.10.4.9.2 choose 201
9.10.4.9.3 pyramid 201
9.10.4.9.4 get_best_k_for_n 202
9.10.4.9.5 max_k_for_n 204
9.10.4.9.6 hamming 204
9.10.4.9.7 normalize 205
9.10.5 Frequency Transforms 205
9.10.5.1 Overview of Frequency Transforms 205
9.10.5.2 do_windowed_imdct 205
9.10.5.3 Functions Supporting do_windowed_imdct 206
9.10.5.3.1 make_window 206
9.10.5.3.2 do_multiple_idct4 206
9.10.5.3.3 do_single_idct4 207
9.10.5.3.4 do_multiple_window 207
9.10.5.3.5 do_single_window 208
9.10.5.4 Band Reshaping 208
9.10.5.5 Haar Transform 209
9.10.5.5.1 Overview of Haar Transform 209
9.10.5.5.2 do_multiple_haar 209
9.10.5.5.3 do_single_haar 210
9.10.5.5.4 haar_coefficient 210
9.10.6 Long Term Synthesis (LTS) 211
9.10.6.1 Overview of LTS 211
9.10.6.2 LTS Constants and Definitions 211
9.10.6.3 do_longterm_synthesis 211
9.10.6.4 Functions Supporting do_longterm_synthesis 212
9.10.6.4.1 lts_make_buffer 212
9.10.6.4.2 lts_set_windows 213
9.10.6.4.3 lts_restore 213
9.10.7 Temporal Hole Fill (informative) 214
9.10.7.1 Overview of Temporal Hole Filling 214
9.10.7.2 do_temporal_hole_fill (informative) 214
9.10.7.3 do_band_temporal_hole_fill (informative) 215
9.10.7.4 random_sign 215
9.10.8 Deemphasis 216
9.10.8.1 Overview of Deemphasis 216
9.10.8.2 do_deemphasis 216
9.10.8.3 Function Called by do_deemphasis 216

9.10.8.3.1 do_channel_deemphasis 216
9.11 LFE Decoding 217
9.11.1 Overview of LFE Decoding 217
9.11.2 LFE Constants and Definition 217
9.11.3 decode_lfe_channel 217
9.11.4 Functions Supporting decode_lfe_channel 218
9.11.4.1 lfe_predict 218
9.11.4.2 lfe_reconstruct 218
9.11.4.3 lfe_synthesize (informative) 218
9.11.4.4 lfe_upsample 219
9.11.4.5 db_to_linear 219

10 Tables and Constants 219
10.1 Overview 219
10.2 System Constants and Tables 220
10.2.1 Basic Constants 220
10.2.2 Coding Modes 220
10.2.3 BandsConfig 220
10.2.3.1 Structure BandsConfig 220
10.2.3.2 Bands Configuration Tables 221
10.2.3.2.1 bands_config_48khz 221
10.2.3.2.2 bands_config_44p1khz 221
10.3 Long Term Synthesis Constants and Tables 222
10.3.1 LTS_FILTER_CODEBOOK 222
10.4 Lognorm Tables and Constants 223
10.4.1 Lognorm Basic Constants 223
10.4.2 LOGNORM_GOLOMB_PARAMS 223
10.4.3 F_PREDICTOR_TABLE 224
10.4.4 B_PREDICTOR_TABLE 225
10.4.5 LOGNORM_REFINEMENT_BOOST_TABLE 225
10.5 Stereo Constants and Tables 225
10.5.1 Stereo Coding Modes 225
10.5.2 Stereo Allocation Modes 226
10.6 Bit Allocation Constants and Tables 226
10.6.1 Bit Allocation Models 226
10.7 Vector Quantizer Constants and Tables 226
10.7.1 Vector Quantizer Basic Constants 226
10.7.2 MAX_BITS_FOR_N 227
10.7.3 MAX_K_FOR_N 227
10.7.4 K_ESTIMATION_COEFFICIENTS 228
10.7.5 BETA_DEQUANT_TABLE 228
10.8 LFE Constants and Tables 246
10.8.1 INTERPOLATION_FILTER_TABLE 246

Annex A (informative): Bibliography 250

Annex B (normative): DTS-UHD ISOBMFF File Encapsulation 251

B.1 Overview 251

B.2 Signalling 251
B.2.1 Track Header 251
B.2.2 Sample Description Box 251
B.2.2.1 Overview of Sample Description Box 251
B.2.2.2 DTSUHDSampleEntry 251
B.2.2.3 DTSUHDSpecificBox 252
B.2.2.3.1 Syntax of DTSUHDSpecificBox 252
B.2.2.3.2 Semantics for DTSUHDSpecificBox 252
B.2.2.3.3 DTSExpansionBox 255

B.3 Storage of DTS-UHD Elementary Streams 255
B.3.1 DTS-UHD Samples 255
B.3.2 Random Access Points 255

Annex C (normative): DTS-UHD in MPEG-2 Transport Streams 257
C.1 Introduction 257
C.2 MPEG-2 System Signalling 257
C.2.1 Buffering Model 257
C.2.2 DTS-UHD Audio Descriptor 257
C.2.2.1 Descriptor Placement 257
C.2.2.2 Descriptor Syntax 257
C.2.2.3 Semantics for Parameters 258
C.3 DTS-UHD PES Packet Encapsulation 262
C.3.1 Stream Type 262
C.3.2 Stream ID 262
C.3.3 Audio Access Unit Alignment in PES Packets 262
C.3.4 Random Access Points 262
Annex D (normative): Delivery of DTS-UHD using MPEG-DASH 263
D.1 Introduction 263
D.2 General considerations 263
D.3 DASH MPD Parameters for DTS-UHD 263
D.4 Support for Preselections 264
D.5 Example MPD Using Preselections 264
Annex E (normative): DTS-UHD CMAF Profiles 266
E.1 Overview 266
E.2 General Guidelines for DTS-UHD CMAF Tracks 266
E.2.1 CMAF Compliance 266
E.2.2 Codec Profiles and Levels 266
E.2.3 Media Access Unit Mapping to Media Samples 266
E.2.4 Media Access Unit Sequence Mapping to CMAF Fragments 266
E.2.5 CMAF Track Constraints for CMAF Switching Sets 266
E.2.6 CMAF Media Profile Internet Media Type 266
E.2.7 CMAF Media Profile Brand 266
E.3 Guidelines for DTS-UHD CMAF Media Profiles 267
E.3.1 General 267
E.3.2 Audio Track Format 267
E.3.3 Loudness and Dynamic Range Control 267
E.3.4 Audio Parameters 267
E.3.5 Audio Presentation Time Adjustment 267
E.4 Delivery Considerations for DTS-UHD CMAF Tracks 267
E.5 Playback Considerations for DTS-UHD CMAF Tracks 268
Annex F (normative): DTS-UHD Media Profiles 269
F.1 Overview of Profiles 269
F.2 DTS-UHD Profile 2 Bitstreams 269
F.2.1 Profile 2 Overview 269
F.2.2 Profile 2 Bitstream Requirements 269
F.2.3 Profile 2 AudioSampleEntry 269
F.3 DTS-UHD Profile 3 Bitstreams 270
F.3.1 Profile 3 Overview 270
F.3.2 Profile 3 Bitstream Requirements 270
F.3.3 Profile 3 AudioSampleEntry 270
F.4 DTS-UHD Device Requirements 270

F.4.1 About DTS-UHD Devices....270
F.4.2 DTS-UHD Profile 2 Device Requirements....270
F.4.3 DTS-UHD Profile 3 Device Requirements....270
F.4.4 DTS-UHD Profile 3 Multi-stream Device Requirements....271

Annex G (informative): Multi-stream Considerations....272

G.1 Introduction....272

G.2 Multi-stream Architecture....272

Annex H (informative): Registrations for DTS-UHD....274

H.1 Overview....274

H.2 MP4RA....274

H.3 IANA....274

History....275

List of Tables

Table 4-1: ucAudPresInterfaceType....28
Table 4-2: Audio Chunk Types....32
Table 5-1: Table of Conversion Constants....37
Table 5-2: ExtractVarLenBitFields....39
Table 5-3: UpdateCode....40
Table 5-4: CountBitsSet_to_1....41
Table 5-5: GetPtrToCurrentByte....41
Table 5-6: GetBitCounter....41
Table 5-7: AdvanceBitReadPointer....41
Table 5-8: CAuPresSelectAPI....42
Table 5-9: Default Presentation....43
Table 5-10: Selected Presentation....43
Table 5-11: Create Presentation from a List of Objects....43
Table 6-1: Frame Metadata Parameters....44
Table 6-2: UnpackMDFrame....44
Table 6-3: ResetMDChunks....46
Table 6-4: Metadata Chunk Types....47
Table 6-5: ExtractAudPresIndex....47
Table 6-6: ExtractMDChunkObjIDList....47
Table 6-7: UpdateExplObjListforAuPres....49
Table 6-8: ExtractandCheckCRC....50
Table 6-9: CRC16_Update4BitsFast....50
Table 6-10: CFrmTblofContent....51
Table 6-11: ExtractFTOC....52
Table 6-12: ExtractStreamParams....53
Table 6-13: Base Duration....55
Table 6-14: Clock Rate....55
Table 6-15: ResolveAudPresParams....56
Table 6-16: ExtractAudioPresParams....57
Table 6-17: ExtractExplicitObjectsLists....59
Table 6-18: ExtractandSaveAllAudPresParams....60
Table 6-19: Audio Chunk Navigation - Class Parameters....61
Table 6-20: ExtractChunkNaviData....61

Table 6-21: ResetList 63
Table 6-22: ResetAudioChunkPresentFlags 63
Table 6-23: FindAudioChunkListIndex 64
Table 6-24: CAudioChunkListHandler::PurgeList 65
Table 6-25: PBR Smoothing Buffer Class Parameters 65
Table 6-26: Extract Peak Bit Rate (PBR) Smoothing Buffer Parameters 65
Table 6-27: SkipReservedandAlignFields 67
Table 7-1: Class Definition of C0x01MDChunkBody 68
Table 7-2: ApplyExplObjList2ObjActMask 69
Table 7-3: ApplyGroupDefaultObjectActivityMask 70
Table 7-4: ExtractandInvQuantParams 71
Table 7-5: ExtractPresScalingParams 73
Table 7-6: Static Metadata Frame Class 74
Table 7-7: ExtractMultiFrameDistribStaticMD 74
Table 7-8: ExtractStaticLandDParams 76
Table 7-9: ExtractLTLMParamSet 79
Table 7-10: Definition of Object Types 80
Table 7-11: Long Term Loudness Measurement Types 80
Table 7-12: DRC Compression Types 81
Table 7-13: DRC Compression Curve Types 81
Table 7-14: ExtractCustomDRCCurves 81
Table 7-15: LookUpDRCCurveParameters 83
Table 7-16: ExtractMetadataForObjects 85
Table 7-17: Reference Screen Aspect Ratios 87
Table 7-18: CheckIfMDIsSuitableforImplObjRenderer 87
Table 7-19: ucRequiredRendererType 88
Table 7-20: ExtractObjectsOrObjGroupDef 90
Table 7-21: ExtractGroupDefinition 91
Table 7-22: ExtractObjectMetadata 92
Table 7-23: Definition of Object Representation Types 96
Table 7-24: ExtractPerObjLTLoudnessMD 97
Table 7-25: ExtractObjectInteractMD 98
Table 7-26: ExtractChMaskParams 100
Table 7-27: unChMask_Index_Table 101
Table 7-28: Mapping Channels to Bits within Channel Activity Mask 102

Table 7-29: GetNumSpeakersInLayout 102
Table 7-30: ExtractRendererConfigParams 103
Table 7-31: m_nBFormatAmbiAdMtrx 105
Table 7-32: ExtractInvQuantMultiUpdtObjMD 111
Table 7-33: InitMultiUpdtObjMD 113
Table 7-34: Extract3DSourceProperties 115
Table 7-35: Renderer Source Types 117
Table 7-36: ExtractREObjectMD 120
Table 7-37: CheckRESetApplicability 122
Table 8-1: Align 124
Table 8-2: Skip 124
Table 8-3: BitsUsed 125
Table 8-4: BitsLeft 125
Table 8-5: BitsTotal 125
Table 8-6: ReadBit 126
Table 8-7: ReadBool 126
Table 8-8: ReadUint 126
Table 8-9: ReadInt 126
Table 8-10: ReadUnary 127
Table 8-11: ReadUniform 127
Table 8-12: ReadGolomb 128
Table 8-13: ReadGolombLimited 128
Table 8-14: ReadGolombWithParams 129
Table 8-15: ReadVLC 129
Table 8-16: ReadLimitsVLC 130
Table 8-17: ReadNonUniformFiveTen 130
Table 8-18: NumBits 130
Table 8-19: Modular 131
Table 8-20: NegativeRiceMapDecode 131
Table 8-21: PositiveRiceMapDecode 131
Table 8-22: DecodeCenterMap 132
Table 8-23: Rand 133
Table 8-24: FixedPoint 133
Table 8-25: Fixed Point Instances 138
Table 9-1: BitStream Data and Methods 138

Table 9-2: read_bitstream....139
Table 9-3: Frame....139
Table 9-4: read_decode_frame....139
Table 9-5: read_padding....140
Table 9-6: FrameHeader....140
Table 9-7: read_frame_header....141
Table 9-8: read_is_sync....142
Table 9-9: read_enable_deemphasis....142
Table 9-10: read_frame_duration....143
Table 9-11: read_sampling_rate....143
Table 9-12: read_num_stream_sets....143
Table 9-13: read_has_padding....143
Table 9-14: StreamSetHeader....144
Table 9-15: read_streamset_header....145
Table 9-16: read_streamset_id....146
Table 9-17: read_stream_information....146
Table 9-18: read_lfe_stream_payload_size....147
Table 9-19: read_mono_stream_payload_size....147
Table 9-20: read_stereo_stream_payload_size....148
Table 9-21: read_stream_bandwidth_mode....148
Table 9-22: AceStreamSet....149
Table 9-23: read_decode_streamset....149
Table 9-24: LFEStream....150
Table 9-25: read_decode_lfe_stream....151
Table 9-26: LFEChannel....151
Table 9-27: read_decode_lfe_channel....152
Table 9-28: read_resolution....153
Table 9-29: read_savings....153
Table 9-30: read_dbnorm....154
Table 9-31: read_predictor....154
Table 9-32: read_lfe_decimated_buffer....154
Table 9-33: Stream....155
Table 9-34: MonoStream....157
Table 9-35: read_decode_mono_stream....157
Table 9-36: StereoStream....158

Table 9-37: read_decode_stereo_stream 158
Table 9-38: read_decode_stream....158
Table 9-39: read_coding_mode....161
Table 9-40: compute_effective_channels....161
Table 9-41: read_long_term_synthesis....161
Table 9-42: read_vq_resolution_mode....162
Table 9-43: read_mdct_mode....162
Table 9-44: read_shf_enabled 163
Table 9-45: read_thf_enabled....163
Table 9-46: read_lognorm_primary 163
Table 9-47: read_num_band_blocks 164
Table 9-48: map_tf_change....165
Table 9-49: undo_differential....165
Table 9-50: enforce_range....166
Table 9-51: read_reconditioning_level....166
Table 9-52: read_allocation....166
Table 9-53: read_num_coded_bands....169
Table 9-54: read_num_bands 169
Table 9-55: read_alloc_mode....169
Table 9-56: read_alloc_model_parameter....170
Table 9-57: read_alloc_model_delta 170
Table 9-58: num_bits_step 171
Table 9-59: read_num_ms_mono_bands 171
Table 9-60: read_stereo_flags 171
Table 9-61: do_allocation_model....173
Table 9-62: do_allocation_model_step 173
Table 9-63: compute_adjustments_from_lognorms....175
Table 9-64: estimate_cost_from_adjustments....175
Table 9-65: read_alloc_diff_steps....176
Table 9-66: remove_guaranteed_bits....176
Table 9-67: compute_delta_alphabets....177
Table 9-68: restore_guaranteed_bits 177
Table 9-69: compute_alloc_from_foundation....178
Table 9-70: compute_first_lognorm_refinement_alloc....178
Table 9-71: adjust_refinements....179

Table 9-72: do_first_lognorm_refinement 180
Table 9-73: do_lognorm_refinement....180
Table 9-74: compute_refinement_correction 181
Table 9-75: transfer_lognorms 181
Table 9-76: do_late_lognorm_refinement 182
Table 9-77: compute_late_lognorm_refinement_alloc....182
Table 9-78: apply_lognorms 183
Table 9-79: Band Reconstruction Constants and Definitions....184
Table 9-80: read_all_bands 184
Table 9-81: distribute_running_balance....186
Table 9-82: read_one_band_mono....186
Table 9-83: save_for_late_refinement....187
Table 9-84: Stereo Constants and Definitions....187
Table 9-85: read_stereo_coding_type 188
Table 9-86: read_one_band_leftright 188
Table 9-87: compute_leftright_allocations....189
Table 9-88: compute_region 190
Table 9-89: read_one_band_midside 190
Table 9-90: midside_to_leftright....191
Table 9-91: VQ Constants and Definitions 192
Table 9-92: read_split_vector....192
Table 9-93: is_split_required....194
Table 9-94: compute_split_parameters 194
Table 9-95: compute_quant_level 195
Table 9-96: read_nonsplit_vector....196
Table 9-97: make_unsigned_pyramid_vector 197
Table 9-98: recondition 197
Table 9-99: make_matrix_4 198
Table 9-100: apply_matrix 199
Table 9-101: flip_vector....200
Table 9-102: interleave_vector....200
Table 9-103: do_spectral_hole_fill....201
Table 9-104: factorial 201
Table 9-105: choose 201
Table 9-106: pyramid 202

Table 9-107: get_best_k_for_n 202
Table 9-108: max_k_for_n 204
Table 9-109: hamming 204
Table 9-110: normalize 205
Table 9-111: do_windowed_imdct 206
Table 9-112: make_window 206
Table 9-113: do_multiple_idct4 207
Table 9-114: do_single_idct4 207
Table 9-115: do_multiple_window 207
Table 9-116: do_single_window 208
Table 9-117: reshape_band 208
Table 9-118: do_multiple_haar 210
Table 9-119: do_single_haar 210
Table 9-120: haar_coefficient 211
Table 9-121: LTS constants 211
Table 9-122: do_longterm_synthesis 212
Table 9-123: lts_make_buffer 212
Table 9-124: lts_set_windows 213
Table 9-125: lts_restore 214
Table 9-126: do_temporal_hole_fill 214
Table 9-127: do_band_temporal_hole_fill 215
Table 9-128: random_sign 215
Table 9-129: do_deemphasis 216
Table 9-130: do_channel_deemphasis 216
Table 9-131: LFE Constants 217
Table 9-132: decode_lfe_channel 217
Table 9-133: lfe_predict 218
Table 9-134: lfe_reconstruct 218
Table 9-135: lfe_synthesize 218
Table 9-136: lfe_upsample 219
Table 9-137: db_to_linear 219
Table 10-1: System Constants 220
Table 10-2: Coding Modes 220
Table 10-3: BandsConfig 220
Table 10-4: bands_config_48khz 221

Table 10-5: bands_config_44p1khz 221
Table 10-6: LTS Filter Codebook 222
Table 10-7: Lognorm Constants 223
Table 10-8: Lognorm Golomb Parameters 223
Table 10-9: F Predictor Table 224
Table 10-10: B Predictor Table 225
Table 10-11: Lognorm Refinement Boost Table 225
Table 10-12: Stereo Coding Modes 225
Table 10-13: Stereo Allocation Modes 226
Table 10-14: Bit Allocation Models 226
Table 10-15: Stereo Allocation Modes 226
Table 10-16: Max Bits for N 227
Table 10-17: Max K for N 227
Table 10-18: K Estimation Coefficients 228
Table 10-19: Beta Dequantization Table 228
Table 10-20: Interpolation_filter_table 246
Table B-1: Defined Audio Formats 251
Table B-2: DTS-UHD Specific Box 252
Table B-3: Frame Duration 252
Table B-4: MaxPayload 253
Table B-5: Speaker Labels for ChannelMask 253
Table B-6: Base Sampling Frequency 254
Table B-7: SampleRateMod 254
Table B-8: Representation Type 254
Table B-9: DTS-UHD SYNCWORD 255
Table C-1: DTS-UHD descriptor 257
Table C-2: Frame Duration 258
Table C-3: MaxPayload 258
Table C-4: Speaker Labels for ChannelMask 260
Table C-5: Base Sampling Frequency 260
Table C-6: SampleRateMod 260
Table C-7: Representation Type 261
Table C-8: DTS-UHD Syncwords 262
Table D-1: DTS-UHD DASH Elements and Attributes 264

List of Figures

Figure 4-1: DTS-UHD Frame Structure....28
Figure 4-2: Default Playback....34
Figure 4-3: Specific Object and Group Selection....35
Figure 4-4: Playback Using Default Settings....35
Figure 4-5: Example of Selecting Playback of Audio Presentation 2....36
Figure 4-6: Example of Selecting Desired Objects to Play Within a Single Stream....36
Figure 6-1: DTS-UHD Stream Structure....67
Figure 7-1: Piece-wise Linear DRC Curve....82
Figure 7-2: Example of Object Groups and 3D Objects....89
Figure 8-1: ACE Decoder....123
Figure 9-1: ACE Elementary Stream....138
Figure G-1: DTS-UHD Multi-stream Example....272

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

Foreword

This Technical Specification (TS) has been produced by Joint Technical Committee (JTC) Broadcast of the European Broadcasting Union (EBU), Comité Européen de Normalisation ELECtrotechnique (CENELEC) and the European Telecommunications Standards Institute (ETSI).

NOTE: The EBU/ETSI JTC Broadcast was established in 1990 to co-ordinate the drafting of standards in the specific field of broadcasting and related fields. Since 1995 the JTC Broadcast became a tripartite body by including in the Memorandum of Understanding also CENELEC, which is responsible for the standardization of radio and television receivers. The EBU is a professional association of broadcasting organizations whose work includes the co-ordination of its members' activities in the technical, legal, programme-making and programme-exchange domains. The EBU has active members in about 60 countries in the European broadcasting area; its headquarters is in Geneva.

European Broadcasting Union
CH-1218 GRAND SACONNEX (Geneva)
Switzerland
Tel: +41 22 717 21 11
Fax: +41 22 717 24 81

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document describes a compressed audio delivery system comprised of a metadata component and a compressed audio component. This system is referred to as DTS-UHD. The stream and metadata structures described here allow efficient delivery of immersive audio. Additionally, the bitstream format of a new audio compression engine is defined.

DTS-UHD supports delivery of Channel-Based Audio (CBA), Object-Based Audio (OBA) and High Order Ambisonic presentations up to the fourth order (HOA Soundfields). This support includes the metadata for signaling content, loudness and dynamics, and the necessary coefficients for rendering the final presentation.

The present document is organized into two main parts, with clauses 4 through 7 describing the DTS-UHD metadata and some system interfaces, which is independent of the coding scheme being used. Clauses 8 through 10 describe the Audio Compression Engine (ACE), which is designed for efficient delivery of compressed audio in broadcast and streaming environments.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference/.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] ETSI TS 102 114: "DTS Coherent Acoustics; Core and Extensions with Additional Profiles".

[2] Recommendation ITU-R BS.1770-4: "Algorithms to measure audio programme loudness and true-peak audio level".

[3] "AmbiX - A Suggested Ambisonics Format", Christian Nachbar; Franz Zotter; Etienne Deleflie; Alois Sontacchi (June 2-3, 2011). Ambisonics Symposium 2011. Lexington (KY).

[4] ISO/IEC 14496-12:2015: "Information technology - Coding of audio-visual objects - Part 12: ISO Base Media File Format".

[5] ISO/IEC 23009-1:2014: "Information technology -- Dynamic adaptive streaming over HTTP (DASH) -- Part 1: Media presentation description and segment formats".

[6] ISO/IEC 23000-19:2018: "Information technology -- Multimedia application format (MPEG-A) -- Part 19: Common media application format (CMAF) for segmented media".

[7] ETSI TS 103 584: "DTS-UHD Point Source Renderer".

[8] ISO/IEC 13818-1:2018: "Information Technology -- Generic coding of moving pictures and associated audio information -- Part 1: Systems".

[9] IETF RFC 6381: "The 'Codecs' and 'Profiles' Parameters for "Bucket" Media Types".

[10] ISO/IEC 60958-1: "Digital audio interface -- Part 1: General".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ATSC Standard A/85: "Techniques for Establishing and Maintaining Audio Loudness for Digital Television".

[i.2] EBU-R128: "Loudness Normalisation and Permitted Maximum Level of Audio Signals".

3 Definition of terms, symbols, abbreviations and document conventions

3.1 Terms

For the purposes of the present document, the following terms apply:

ACE channel: one waveform within an ACE Stream

ACE frame: one ACE access unit

ACE group of frames: contiguous sequence of frames that start with a Sync Frame

ACE sync frame: ACE frame that may also serve as a random access point

ACE stream: subset of an ACE frame containing one or two channels

ACE streamset: logical group of streams within an ACE Frame

NOTE: A typical example of a streamset is the collection of streams making up a traditional 5.1 or 7.1 bed. The composition of a streamset is static from frame to frame.

audio object: artistic component of the audio signal comprised of one or more elements such as music, effects, speech or a bed

NOTE: An audio object may consist of one or more channels of audio.

audio presentation: selected collection of Channels, or Objects and Object Groups used together to generate the rendered output

audio stream: sequence of synchronized audio frames

band: collection of contiguous frequency bins corresponding to perceptual grouping of frequency bins according to a Bark scale

NOTE: For ACE coding, Bands are numbered from 0 to 21 (22 Bands in total).

bed: channel-based mix to which audio objects will be added to create a complete presentation

coding mode: type of compression used on the audio data

DTS-UHD Audio Frame: one DTS-UHD access unit

NOTE: DTS-UHD Audio frames are either Sync Frames or Non-Sync Frames. Only Sync Frames may serve as an SAP (Stream Access Point). DTS-UHD Sync Frames will contain an ACE Sync Frame.

frame: access unit

frame duration: period of time represented in one audio frame

NOTE: Duration may be represented in audio samples per channel at a specific audio sampling frequency or in seconds.

non-sync frame: frame which is not a random access point

object group: selected collection of audio objects to be played together

sample resolution: number of bits used to represent a sample value

sampling rate: temporal frequency at which samples are presented, a.k.a. sampling frequency

SPDIF: S/PDIF or TOSLINK serial audio interfaces

sync frame: frame that may be a random access point

XLL: lossless audio compression engine defined in ETSI TS 102 114 [1]

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ACE	Audio Compression Engine
ACN	Ambisonic Channel Number
AM	Adapter Matrix
API	Application Program Interface
ATSC	Advanced Television Systems Committee
CBA	Channel-Based Audio
CLR	Center Left Right
CRC	Cycle Redundancy Check
DASH	Dynamic Adaptive Streaming over HTTP
DRC	Dynamic Range Control
DWORD	Double Word
EOL	Explicit List of Objects
FTOC	Frame Table of Contents
GoF	Group of Frames
HDTV	High Definition TeleVision
HLS	HTTP Live Streaming
HOA	Higher Order Ambisonics
IIR	Infinite Impulse Response
IMDCT	Inverse Modified Discrete Cosine Transform
LFE	Low Frequency Effects
L-PCM	Linear Pulse Code Modulation
LR	Left Right
LSB	Least Significant Bit
LTS	Long Term Synthesis
MD	Metadata (Specifically referring to a Metadata Chunk)
MDCT	Modified Discrete Cosine Transform
MFD	Multi-Frame Distributed
MSB	Most Significant Bit
OBA	Object-Based Audio
OTT	Over the top
PBR	Peak Bit Rate
PCM	Pulse Code Modulation
PES	Packetized Elementary Stream

PID	Package IDentifier
PMT	Program Map Table
RE	Rendering Exception
REM	Rendering Exception Matrix
SHF	Spectral Hole Filling
THF	Temporal Hole Filling
VBR	Variable Bit Rate
VQ	Vector Quantization

3.4 Document Conventions

A number of conventions are applied throughout the present document:

- In parameter descriptions, most significant byte and most significant bit first (big endian), convention is utilized unless otherwise specified.
- The existence of many bit fields is determined by conditions of prior bits in the stream. As such, in many cases the bitstream elements are described using standard 'C' conventions, with a pseudo function **`ExtractBits()`** representing the bit field of interest for that description.
- Bit field descriptions are described in presentation order.
- In many cases, variable names are assigned to fields as they are being described. In some cases, the variable may be modified during definition, such as:

```
nuFieldSize = ExtractBits(4)+1
```

- **`ByteAlign`** or **`Align`**() fields are used throughout the present document to insert up to 7 padding bits in order to force the next defined parameter to begin on a byte boundary. In cases where the stream happens to already be aligned to a byte boundary, then no bits are added. An encoder/stream packager will nominally set padding bits to 0. The decoder (or receiver) should make no assumption regarding value of the padding bits.

4 DTS-UHD System Overview

4.1 Overview

The DTS-UHD coding system is the third generation of DTS audio delivery formats. It is designed to both improve efficiency and deliver a richer set of features than the second generation DTS system.

The first two generations of DTS codecs were designed primarily for Channel-Based Audio (CBA). A primary advantage of CBA is a relatively light metadata burden, as a stream is constrained to a very limited number of playback options.

DTS-UHD is primarily designed to support audio objects, where a given object can represent a channel-based presentation, an Ambisonic sound field or audio objects used in Object-Based Audio (OBA). There are at least two major advantages to Object-Based Audio:

- Adaptability to the listening environment. Audio programs mixed using OBA do not need to assume a particular listening environment (e.g. speaker layout). This allows the playback system to render the best experience possible relative to that room.
- The ability to adapt to the listener's preference. OBA allows efficient support for features like alternate speech tracks and listener customizations such as changing the speech volume (without affecting anything else).

The DTS-UHD format can support up to 224 discrete audio Objects for OBA and 32 Object Groups in one stream.

A given DTS-UHD decoder is characterized by its "lane count". An audio object can consist of multiple streams (or "channels"). The lane count of a presentation would be the sum of the channels of the objects being presented. For example, if a presentation consists of a 5.1 channel Music and Effects track and one stereo speech track, the lane count for the presentation would be seven, (the LFE is not counted when assessing the lane count of a presentation).

Nominally, each DTS-UHD stream will include a presentation that can be played on the minimum player configuration, (based on the version number), that will be expected to play that stream. Such product definitions are not within the scope of the present document, but are a consideration when considering a DTS-UHD ecosystem.

One of the challenges of OBA is the additional metadata necessary to support a presentation. DTS-UHD has provisions for reducing the frequency at which metadata is repeated, thus reducing this burden. OTT streaming methods such as DASH and HLS utilize larger media chunks in Fragments that have guaranteed entry points. DTS-UHD permits encoding options to only update metadata when necessary.

DTS-UHD can support XLL (lossless audio compression) introduced in ETSI TS 102 114 [1] and ACE (lossy compression engine), introduced in the present document starting in clause 8.

4.2 Stream Construction

4.2.1 Construction of a DTS-UHD Audio Frame

The DTS-UHD stream is a sequence of DTS-UHD frames. The DTS-UHD frames consist of three major construction elements:

1) Frame Table Of Contents (FTOC) - This element allows a decoder to navigate directly to elements of interest with the frame.

2) Metadata Chunk elements.

3) Audio Chunk elements.

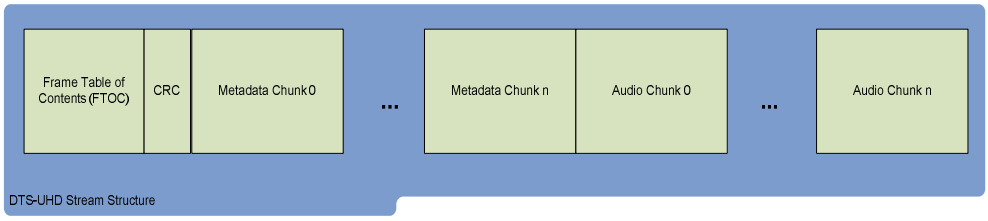

Figure 4-1: DTS-UHD Frame Structure

4.2.2 Frame Table of Contents (FTOC)

The main components of the FTOC are the Sync Word (the first 4 bytes of the FTOC), which indicates whether the frame is a sync frame or non-sync frame, default presentations and dependency information, and the navigation tables to the metadata chunks and audio chunks being delivered in the frame payload.

The FTOC is the only component that is guaranteed to be present in every DTS-UHD frame.

A presentation mask for each presentation is stored in the FTOC that indicates which objects are to be played. The default dependent audio presentation mask defines the default program for playback. Other programs may also be present in the FTOC.

A player may invoke the default program, select another program (if any are available), or provide a custom playback configuration (e.g. based on a manifest from a media presentation descriptor or configuration file). The tokens used to invoke the decoder in the player, (the token is stored in *ucAudPresInterfaceType*), are shown in Table 4-1.

Table 4-1: ucAudPresInterfaceType

Value	Mnemonic
0	API_PRES_SELECT_DEFAULT_AP
1	API_PRES_SELECT_SPECIFIC_AP
2	API_PRES_SELECT_OBJECT_ID_LIST

When *m_ucAudPresInterfaceType* is either API_PRES_SELECT_DEFAULT_AP or API_PRES_SELECT_SPECIFIC_AP the *unDepAuPresMask* is extracted from the bitstream and saved for the selected audio presentation. This mask indicates all objects that need to be played in order to create the desired audio playback presentation. For each dependent audio presentation either:

- the default active objects (according to their activity flags) are played; or
- the objects from an explicitly defined list of active object (transmitted in the stream) are played. The list is stored as a bit mask for each dependent presentation (*m_unDepAuPresExplObjListMask[]*).

When *m_ucAudPresInterfaceType* is API_PRES_SELECT_OBJECT_ID_LIST, it is not known yet which of the audio presentations contain objects from the list. Consequently all audio presentations in the stream shall be considered for playback and the corresponding bits within *unDepAuPresMask* are all set to "1". The masks transmitted in the stream are ignored.

4.2.3 Sync Frames

The decoder does not need any information from previous or future frames to produce a frame of output L-PCM samples from a sync frame. All parameters necessary to unpack metadata and audio chunks, describe audio chunks, render and process audio samples and generate a frame of L-PCM samples can be found within the payload of a sync frame. A decoder can attempt to establish initial synchronization only in a Sync frame. These frames represent the random access points for random navigation to a particular location in the bitstream.

4.2.4 Non-sync Frames (Predictive Frames)

A non-sync frame permits both metadata chunks and audio chunks to minimize payload size by only sending parameters that have changed in value since the previous frame or sync frame. All parameters that are not re-transmitted are assumed to maintain their previous value. Any value or set of values may be updated in a non-sync frame.

A decoder shall not attempt to establish initial synchronization using non-sync frames, nor may these non-sync frames be used as random access points.

4.3 Carriage of Metadata

4.3.1 Organization of Metadata

Metadata for multiple objects and object groups is packed together within an associated metadata chunk. Each chunk of a particular type has its associated audio presentation index. Notice that chunks of different metadata chunk types may have the same audio presentation index; however, two metadata chunks of the same type will have different audio presentations indexes. An example of two different metadata chunk types associated with the same audio presentation would be:

- one metadata chunk type is defined to carry object and object group metadata;
- other metadata chunk types may be defined for describing post-processing parameters that are applied by downstream processors, such as a binaural renderer.

Since both of these chunks together describe the audio presentation that is being played they have the same associated audio presentation index.

4.3.2 Metadata Chunks

Metadata chunks carry the full description of the audio data chunks and how decoded audio shall be rendered for final audio presentation. Additional types of metadata that may be useful for categorization of an audio presentation, support of some post-processing functionality, etc., may also be carried within the metadata chunks.

Each DTS-UHD stream decoder instance can be configured from the system layer in three different ways, depending on the type of information that is provided, in order to select desired audio playback presentation. In particular, within DTS-UHD metadata frame:

- Metadata describing different audio presentations may be present.
- The list of audio presentations/audio objects to be decoded within this stream is passed through a decoder instance API by means of:
  - Enable/disable default audio presentation flag (*bEnblDefaultAuPres*). In this case the audio presentation with the lowest presentation index (i.e. *ucAudPresIndex* = 0) will be selected, and the default objects within that presentation will be played (if *bEnblDefaultAuPres* is TRUE) or outputs will be muted (if *bEnblDefaultAuPres* is FALSE). This case is indicated by *m_ucAudPresInterfaceType* = API_PRES_SELECT_DEFAULT_AP.
  - Desired audio presentation index (*ucDesiredAuPresIndex*). In this case the explicitly selected desired audio presentation and the default objects within that presentation will be played. This API is aware only of the selectable audio presentations and non-selectable audio presentations are not counted in *ucDesiredAuPresIndex*. This case is indicated by *m_ucAudPresInterfaceType* = API_PRES_SELECT_SPECIFIC_AP.
  - Explicit list of desired object IDs to be played (*unDesiredObjIDList*). In this case only the audio presentations containing objects from this list are unpacked and only the objects from the list are played. The default activity flags for all other objects are considered to be FALSE. If some of the listed object IDs are that of an object group, then that group's object activity mask is respected and the corresponding referenced objects are played. This case is indicated by *m_ucAudPresInterfaceType* = API_PRES_SELECT_OBJECT_ID_LIST.
- The speaker layout of the listening environment is provided by the system layer through a decoder API by means of the bit-mask *unOutChMask.*
- Prior to extraction of any metadata chunks the corresponding metadata chunk CRC is checked for validity of received data.
- Based on the *unDepAuPresMask* (constructed within FTOC) only the metadata chunks that have a matching audio presentation index are extracted from the stream; all other metadata chunks are ignored and skipped over.
- Pseudo-code listed in clause 6.2 only recognizes metadata chunks with Chunk ID equal to 0x01, 0x02, 0x03 and 0x04.
- For metadata chunks of type 0x01:
  - Immediately after the extraction of audio presentation index, the list of object IDs described in this metadata chunk is extracted.
  - If the *m_AuPrSelectAPI_SampledatSyncFrm.m_ucAudPresInterfaceType* is API_PRES_SELECT_OBJECT_ID_LIST the extracted object list is compared to the desired object list. If any of the object IDs from the desired object ID list are found in the extracted object ID list this metadata chunk is pronounced to be active. The explicit list of active objects (*m_unDepAuPresExplObjListMask[]*), is updated to reflect the desired active objects.
  - Memory for every active 0x01 metadata chunk is allocated dynamically:
    - After the extraction of an active 0x01 metadata chunk the flags that indicate each object's activity are updated to obey the list of explicitly defined active objects (*m_unDepAuPresExplObjListMask[]*).

- For 0x02, 0x03 and 0x04 chunks, the decoder exports the pointers, (through an API), within the stream buffer corresponding to the beginning of each of the chunks and the chunk sizes.

In every sync frame all active metadata is transmitted and all previous states are reset (*ResetMDChunks()*) with the exception of static metadata (pointed to by *m_pCMFDStaticMD*). When static metadata is distributed over multiple frames, it will be completely refreshed from one sync frame to the next. For example, if the interval between consecutive sync frames is 10 frame periods, then (conceivably) as little as 1/10 of the static metadata could be sent in each frame.

4.3.3 Fundamental Components of the Metadata Chunk

4.3.3.1 Metadata Chunk: Data

The data field contains the data of the chunk. The format of the data is specific to the type of chunk. Currently (as of the present document) only *ChunkType* = 0x01 is defined. The details of this chunk are in clause 7.

4.3.3.2 Reserved and Byte Align Fields

Reserved fields and fields for padding to the next byte boundary are defined at the end of each metadata chunk. These fields are used to insure byte alignment for the following data structure (e.g. another metadata chunk), and for extending the metadata chunk with new data structures. The decoder shall always assume that these fields are present and navigate past those fields based on the chunk size information.

4.3.3.3 Metadata Chunk CRC Word

If the CRC flag (transmitted within FTOC: Metadata and Audio Chunk Navigation Parameters) corresponding to particular metadata chunk is TRUE, the metadata chunk CRC (16-bit) word is transmitted in order to allow verification of the metadata chunk data. This CRC value is calculated over metadata fields, starting from and including the MD Chunk ID and up to and including the byte alignment field prior to the CRC word (*unMDChunkSize*-2 bytes in total).

The decoder shall:

- Calculate the CRC(16) value over the metadata chunk data fields i.e. over *unMDChunkSize* - 2 bytes (excluding CRC itself) of data.
- Extract the 16-bit MD chunk CRC field and compare it against the calculated CRC(16) value.
- If the two values match, reverse back to the beginning of metadata chunk (return TRUE); otherwise, pronounce data corruption (return FALSE).

4.4 Audio Chunks

Audio chunks carry the compressed audio samples. Audio samples may be representing speaker feeds, waveforms associated with a 3D audio object, waveforms associated with a sound field audio representation, or some other valid audio representation. The associated metadata chunk fully describes the way a particular audio chunk is presented and the type of audio carried within each audio chunk.

The metadata chunks and audio chunks are packed immediately after the FTOC CRC word as shown in Figure 4-1.

Nominally, an audio chunk points to a minimum collection of compressed waveforms that can be decoded without dependency on any other audio chunks. All compressed waveforms within an audio chunk that has been selected for decoding shall be decoded and played together. In some cases an elementary stream may already have its own sub-division into individually decodable parts in which case all encoded objects within one DTS-UHD stream can be packed into a single audio chunk (i.e. 0x01, 0x03 or 0x04 audio chunk). In some cases (i.e. 0x40 audio chunk) an audio chunk does not point to any compressed waveforms but rather it points to header/metadata information within a compressed audio elementary stream.

Organization of the audio parameters into chunks allows for the addition of new features and/or quality improvements by simply defining new chunks. When the new audio chunks are defined, the old decoders shall recognize and extract audio chunks that they are aware of (subject to the system constraints like the maximum number of decodable channels, the maximum sampling frequency, etc.) and ignore all other audio data chunks.

For each audio chunk:

- The chunk ID, the payload size in bytes and the audio chunk index are all transmitted within the FTOC payload.
- There are no header parameters in addition to the chunk payload.
- An audio chunk type, as pointed by the chunk ID, identifies the type of data stored in a corresponding audio chunk.

The list of currently defined audio chunks is shown in Table 4-2.

Table 4-2: Audio Chunk Types

Audio Chunk Type	AudioChunk ID	Interpretation of m_ucObjNaviWitinACIndex (see clause 7.8.11.7)
NULL Chunk	0x00	
ACE frame payload consisting of encoded audio data for single or multiple channels/objects. It includes DTS ACE frame headers. This audio chunk type cannot coexist with either 0x03 or 0x04 audio chunk types within the same DTS-UHD stream.	0x01	Corresponds to the ACE stream set index.
DTS ACE channel-set payload consisting of encoded audio data for single or multiple channels/objects. Does NOT include DTS ACE frame header but it relies on frame header parameters transmitted in associated 0x01 audio chunk. This audio chunk type cannot coexist with either 0x03 or 0x04 audio chunk types within the same DTS-UHD stream.	0x02	
Concatenation of two DTS ACE frame payloads packed within the single audio chunk. It is used when the metadata frame duration (in time) is twice the duration of the DTS ACE frame. This audio chunk type cannot coexist with either 0x01, 0x02 or 0x04 audio chunk types within the same DTS-UHD stream.	0x03	
Concatenation of four DTS ACE frame payloads packed within the single audio chunk. It is used when the metadata frame duration (in time) is four times the duration of the DTS ACE frame. This audio chunk type cannot coexist with either 0x01, 0x02 or 0x03 audio chunk types within the same DTS-UHD stream.	0x04	
Reserved	0x05 - 0x3F	
XLL Frame Header Data. This chunk is persistent and usually associated with appearing and disappearing audio chunks of type 0x41 and/or 0x42.1.	0x40	
XLL Channel Set Header and Channel Set Data for all segments and frequency bands. The navigation table for segments and bands within this channel set shall be included at the end of this audio chunk.	0x41	Corresponds to the channel index within the XLL channel set that is associated with the first waveform of an object. It is assumed that if an object is associated with multiple waveforms the audio data corresponding to these waveforms is packed consecutively within the XLL audio chunk.
Collection of two channel sets each consisting of: • XLL Channel Set Header; and • XLL Channel Set Data for all segments and frequency bands. These two channel sets are required to coexist and shall be decoded together. The navigation table for segments and bands of both channel sets shall be included at the end of this audio chunk.	0x42	

Audio Chunk Type	AudioChunk ID	Interpretation of m_ucObjNaviWitinACIndex (see clause 7.8.11.7)
Typical use of this audio chunk is for an object that has between 17 and 32 PCM waveforms to be compressed (XLL channel set has an upper limit of 16 channels and the channels of the second channel set shall be counted as the continuation of the channel list from the first channel set).		
Reserved	0x43 - 0xFF	
NOTE: If present, the first instance of this audio chunk type is the first audio chunk.		

4.5 Organization of Streams

4.5.1 Objects, Object Groups, Presentations

The fundamental unit of a DTS-UHD stream is the object. The simplest example of a DTS-UHD stream would be a stream containing one object. For example, one stereo audio presentation, or even a single 5.1 or 7.1 channel presentation, could be handled in such a manner.

Object Groups provide a mechanism to associate objects that should always be used together with a single identifier.

Presentations are composed of a selection of objects and/or object groups. Membership of an object or object group in a presentation is non-exclusive.

4.5.2 Properties of Objects

The **object** metadata carries parameters needed to:

- Uniquely identify an object within a DTS-UHD stream.
- Point to associated audio waveforms.
- Describe the audio object properties necessary to render associated audio waveforms.
- Assign whether the Default Playback status of the object is Active or Silent.
- Describe the type of audio content the object is associated with.
- Describe the object's loudness and dynamics properties.

4.5.3 Properties of Object Groups

The **object group** metadata carries parameters needed to:

- Uniquely identify an object group within one DTS-UHD stream by means of a unique object ID.
- Indicate which objects belong to the group by means of a list of object IDs.
- Assign whether the default status is to be played or to be silent.
- Indicate which objects within the group by default shall be rendered and which objects shall be silent; note that the object group setting can overwrite the individual object default activity flags.

Note that object groups do not directly point to any audio waveforms but only point to the specific object IDs. The definition of object groups is fairly generic and hence can be used for almost arbitrary object grouping.

4.5.4 Audio Presentations

Multiple audio presentations may be defined within a single DTS-UHD bitstream. Each audio presentation has unique audio presentation index within a stream and may be categorized as:

- **selectable:** where the playback of this presentation does not need playback of any of the audio presentations with the higher audio presentation index; or
- **non-selectable:** where the playback of this presentation requires playback of one or more audio presentations with the higher presentation index. Note that the non-selectable presentation is the default presentation if it is the only audio presentation in the stream.

For each selectable audio presentation:

- playback of some dependent audio presentations with the lower audio presentation index may be required;
- required lower indexed presentations are indicated by a dependent presentation mask:
  - if a specific list of objects is to be played within a dependent audio presentation, an explicit list of objects (EOL) is transmitted within the FTOC metadata;
  - if within a dependent audio presentation, only the default active objects are to be played. The EOL is not transmitted;
- scaling parameters specific to an audio presentation may be transmitted in the stream;
- a long term loudness measurement parameter set may be transmitted in the stream that is specific to an audio presentation.

Figure 4-2 and Figure 4-3 show two playback examples, the first one using Default Playback and the second one using specific object and object group selection. In both examples, the darker blocks indicate the active elements.

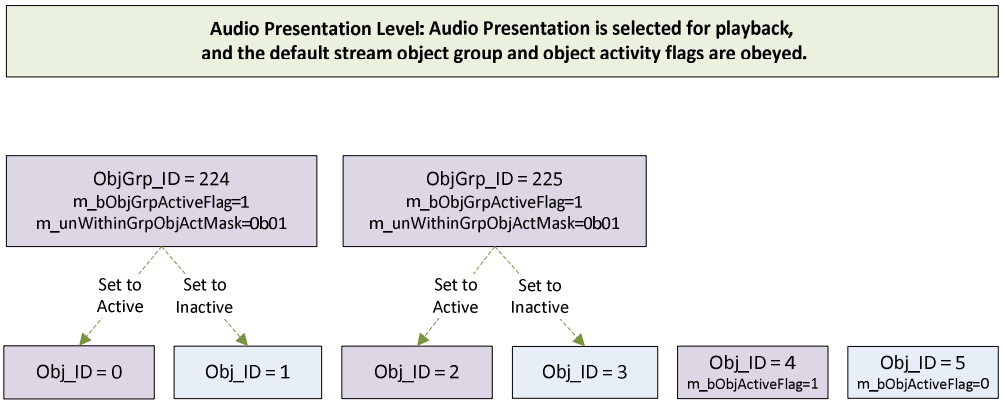

Figure 4-2: Default Playback

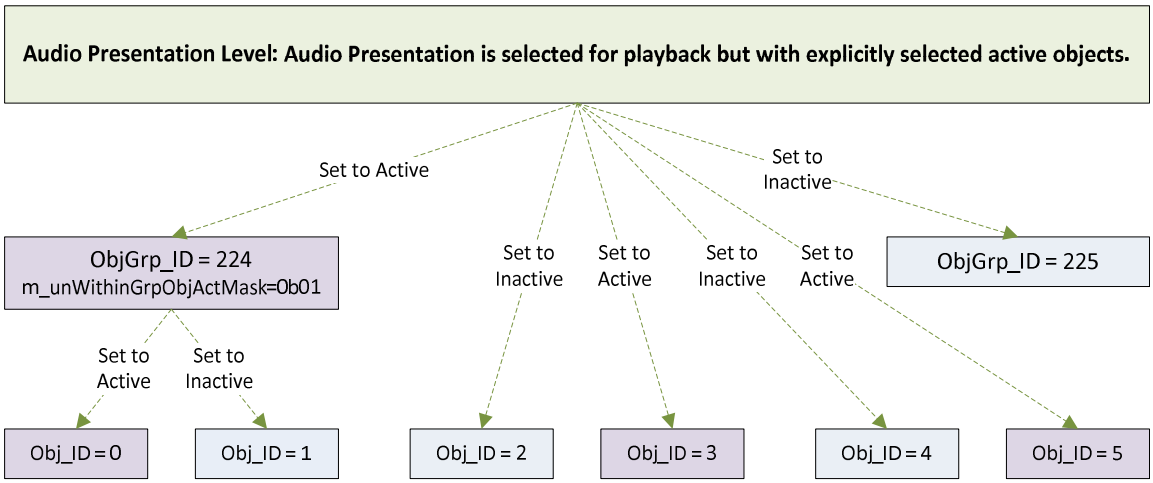

Figure 4-3: Specific Object and Group Selection

Each DTS-UHD stream requires a dedicated DTS-UHD stream decoder instance. Each DTS-UHD stream decoder instance is configured with one of the three types of audio presentation selection APIs:

- Play the default presentation only
- Play a selected presentation
- Play a list of selected elements (objects)

Once configured, the particular instantiation of a stream decoder cannot change the type of presentation selection API.

The following three diagrams illustrate examples of selecting desired objects to play from multiple presentations within a single stream. Purple blocks indicate active audio presentations and corresponding object groups and objects.

Figure 4-4 is an example of playback using the default audio presentation (*m_bEnblDefaultAuPres*=1), i.e. the lowest indexed selectable audio presentation (AP1). The default object group and object activity flags within AP1 are being obeyed. In addition external object groups are activated.

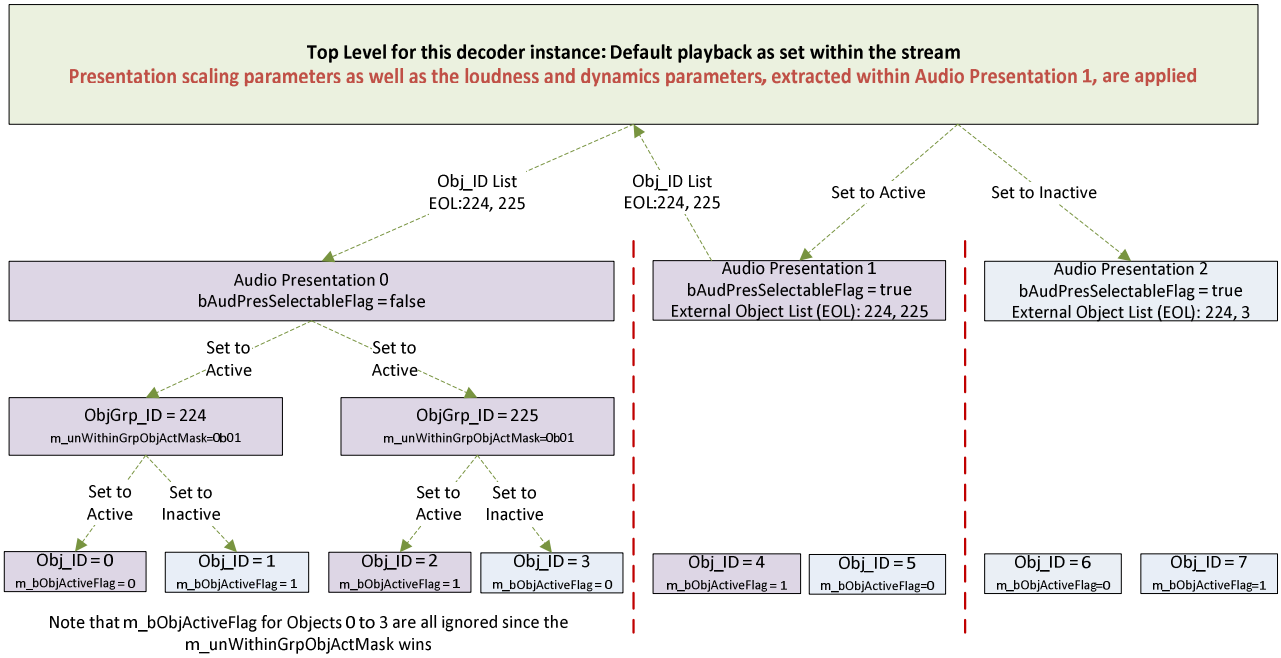

Figure 4-4: Playback Using Default Settings

The diagram in Figure 4-5 shows default object group and object activity flags within AP2 are being obeyed. In addition external object groups are activated.

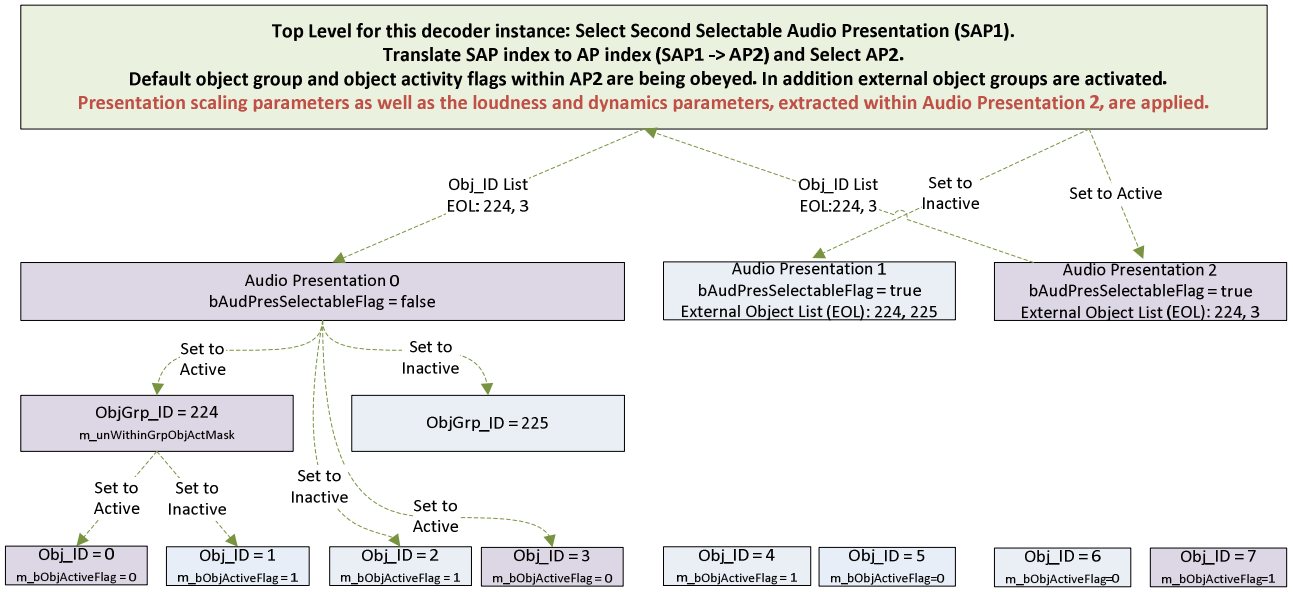

Figure 4-5: Example of Selecting Playback of Audio Presentation 2

The diagram in Figure 4-6 shows no default presentations being selected; rather an explicit playlist is selected which can override all defaults.

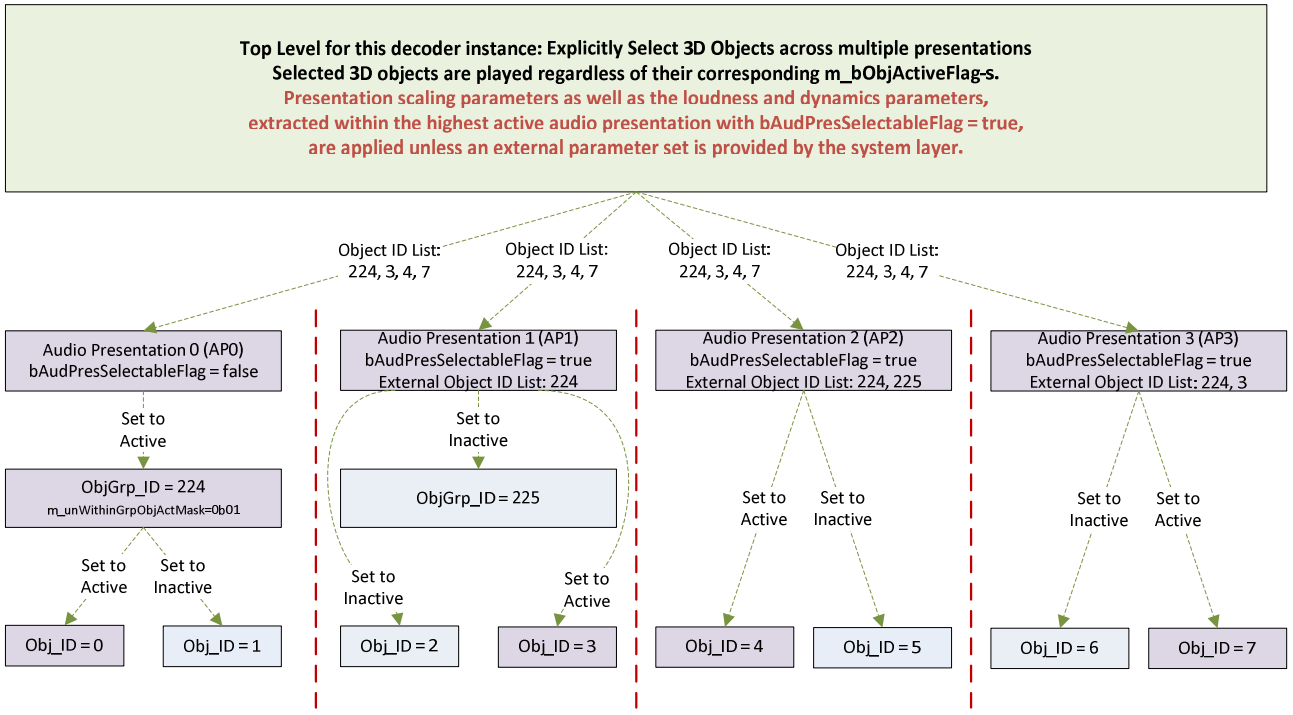

Figure 4-6: Example of Selecting Desired Objects to Play Within a Single Stream

Each DTS-UHD stream is encoded independently, i.e. without any awareness of any objects within any other DTS-UHD streams. Consequently, when multiple DTS-UHD streams are to be played together, it is expected that a system layer will make an appropriate audio presentation/object list selection for each of the involved DTS-UHD streams. The system layer will make these selections based on some manifest file that describes available choices within each of the streams.

5 DTS-UHD Header Tables and Helper Functions

5.1 Overview

This clause includes conversion constants and parameter tables to assist the description and interpretation of the metadata. Additionally, some essential APIs are described for initiating a decoder session.

5.2 Constants, Tables and Helper Functions

5.2.1 Fixed Point Constants

Table 5-1 defines some Q format number constants used in rational number calculations.

Table 5-1: Table of Conversion Constants

Mnemonic	Value	Meaning
FOURTEEEN_DB_LIN_Q15	180613	10^(14/20)* (1 << 15) + (1 << 14)
ONE_IN_Q15	32768	1 << 15
SQRT2Q1DOT15	62725	√2 × (1 << 15) + (1 << 14)

5.2.2 Look-up Tables

5.2.2.1 Scale Factor Table

```
const uint unScaleFactorsTable[] =
{
                0,      41,     52,     65,     82,     104,    130,    164,
                207,    260,    328,    413,    519,    654,    823,    1036,
                1163,   1305,   1464,   1642,   1843,   2068,   2320,   2603,
                2920,   3277,   3677,   4125,   4629,   5193,   5827,   6172,
                6538,   6925,   7336,   7771,   8231,   8719,   9235,   9783,
                10362,  10976,  11627,  12315,  13045,  13818,  14637,  15504,
                16423,  17396,  18427,  19519,  20675,  21900,  23170,  24573,
                26029,  27571,  29205,  30935,  32768
};
```

5.2.2.2 Long Term Loudness Measure Table

```
const float rLongTermLoudnessMeasure_Table[] =
{
 -40.0000f,   -39.0000f,   -38.0000f,   -37.0000f,   -36.0000f,   -35.0000f,   -34.0000f,   -33.0000f,
 -32.0000f,   -31.0000f,   -30.0000f,   -29.5000f,   -29.0000f,   -28.5000f,   -28.0000f,   -27.5000f,
 -27.0000f,   -26.7500f,   -26.5000f,   -26.2500f,   -26.0000f,   -25.7500f,   -25.5000f,   -25.2500f,
 -25.0000f,   -24.7500f,   -24.5000f,   -24.2500f,   -24.0000f,   -23.7500f,   -23.5000f,   -23.2500f,
 -23.0000f,   -22.7500f,   -22.5000f,   -22.2500f,   -22.0000f,   -21.7500f,   -21.5000f,   -21.2500f,
 -21.0000f,   -20.5000f,   -20.0000f,   -19.5000f,   -19.0000f,   -18.0000f,   -17.0000f,   -16.0000f,
 -15.0000f,   -14.0000f,   -13.0000f,   -12.0000f,   -11.0000f,   -10.0000f,   -9.0000f,    -8.0000f,
 -7.0000f,    -6.0000f,    -5.0000f,    -4.0000f,    -3.0000f,    -2.0000f,    -1.0000f,     0.0f
};
```

5.2.2.3 Per-Object Long Term Loudness Measure Table

Entries in the *rPerObjLongTermLoudnessMeasure_Table[]* represent per-object long term loudness measures in dB relative to full scale.

```
const float rPerObjLongTermLoudnessMeasure_Table[] =
{
 -61.0000f,   -58.0000f,   -55.0000f,   -52.0000f,   -49.0000f,   -46.0000f,   -44.0000f,   -42.0000f,
 -40.0000f,   -39.0000f,   -38.0000f,   -37.0000f,   -36.0000f,   -35.0000f,   -34.0000f,   -33.0000f,
 -32.0000f,   -31.0000f,   -30.0000f,   -29.5000f,   -29.0000f,   -28.5000f,   -28.0000f,   -27.5000f,
 -27.0000f,   -26.7500f,   -26.5000f,   -26.2500f,   -26.0000f,   -25.7500f,   -25.5000f,   -25.2500f,
 -25.0000f,   -24.7500f,   -24.5000f,   -24.2500f,   -24.0000f,   -23.7500f,   -23.5000f,   -23.2500f,
```

```
 -23.0000f,   -22.7500f,   -22.5000f,   -22.2500f,   -22.0000f,   -21.7500f,   -21.5000f,   -21.2500f,
 -21.0000f,   -20.5000f,   -20.0000f,   -19.5000f,   -19.0000f,   -18.0000f,   -17.0000f,   -16.0000f,
 -15.0000f,   -14.0000f,   -13.0000f,   -11.0000f,   -9.0000f,    -6.0000f,    -3.0000f,     0.0f
};
```

5.2.2.4 Quantization Table for DRC Fast Attack Smoothing Constant

The extracted 6-bit index is mapped to 64 values in the range [0 to 25] representing the value of DRC fast attack smoothing constant.

```
const float rDRCFastAttackTable[] =
{
     0.0000f,    0.2000f,    0.4000f,    0.6000f,    0.8000f,    1.0000f,    1.2000f,    1.4000f,
     1.6000f,    1.8000f,    2.0000f,    2.2000f,    2.4000f,    2.6000f,    2.8000f,    3.0000f,
     3.2000f,    3.4000f,    3.6000f,    3.8000f,    4.0000f,    4.2000f,    4.4000f,    4.6000f,
     4.8000f,    5.0000f,    5.2000f,    5.4000f,    5.6000f,    5.8000f,    6.0000f,    6.2000f,
     6.4000f,    6.6000f,    6.8000f,    7.0000f,    7.2000f,    7.4000f,    7.6000f,    7.8000f,
     8.0000f,    8.5000f,    9.0000f,    9.5000f,    10.0000f,   10.5000f,   11.0000f,   11.5000f,
     12.0000f,   12.5000f,   13.0000f,   13.5000f,   14.0000f,   15.0000f,   16.0000f,   17.0000f,
     18.0000f,   9.0000f,    20.0000f,   21.0000f,   22.0000f,   23.0000f,   24.0000f,   25.0000f
};
```

5.2.2.5 Quantization Table for DRC Fast Release Smoothing Constant

The extracted 6-bit index is mapped to 64 values in the range [0 to 1 000] representing the DRC fast release smoothing constant.

```
const float rDRCFastReleaseTable[] =
{
             0.0f,     5.0f,     10.0f,    15.0f,    20.0f,    25.0f,    30.0f,    35.0f,
             40.0f,    45.0f,    50.0f,    55.0f,    60.0f,    65.0f,    70.0f,    75.0f,
             80.0f,    85.0f,    90.0f,    95.0f,    100.0f,   110.0f,   120.0f,   130.0f,
             140.0f,   150.0f,   160.0f,   170.0f,   180.0f,   190.0f,   200.0f,   210.0f,
             220.0f,   230.0f,   240.0f,   250.0f,   260.0f,   270.0f,   280.0f,   290.0f,
             300.0f,   325.0f,   350.0f,   375.0f,   400.0f,   425.0f,   450.0f,   475.0f,
             500.0f,   525.0f,   550.0f,   575.0f,   600.0f,   625.0f,   650.0f,   675.0f,
             700.0f,   725.0f,   750.0f,   800.0f,   850.0f,   900.0f,   950.0f,   1000.0f
};
```

5.2.2.6 Quantization Table for DRC Slow to Fast Threshold

Entries in the *rDRCSlow2FastThrshld_Table[]* represent the threshold values expressed in the power domain in range [1 to 0,001].

```
    const float rDRCSlow2FastThrshld_Table[] =
    {
1.0000000f,  0.8912509f,  0.7943282f,  0.7079458f,  0.6309574f,   0.5623413f,  0.5011872f,  0.4731513f,
0.4466836f,  0.4216965f,  0.3981072f,  0.3758374f,  0.3548134f,   0.3349654f,  0.3162278f,  0.2985383f,
0.2818383f,  0.2660725f,  0.2511886f,  0.2371374f,  0.2238721f,   0.2113489f,  0.1995262f,  0.1883649f,
0.1778279f,  0.1678804f,  0.1584893f,  0.1496236f,  0.1412538f,   0.1333521f,  0.1258925f,  0.1188502f,
0.1122018f,  0.1000000f,  0.0891251f,  0.0794328f,  0.0707946f,   0.0630957f,  0.0562341f,  0.0501187f,
0.0446684f,  0.0398107f,  0.0354813f,  0.0316228f,  0.0281838f,   0.0251189f,  0.0223872f,  0.0199526f,
0.0177828f,  0.0158489f,  0.0141254f,  0.0125893f,  0.0112202f,   0.0100000f,  0.0079433f,  0.0063096f,
0.0050119f,  0.0039811f,  0.0031623f,  0.0025119f,  0.0019953f,   0.0015849f,  0.0012589f,  0.0010000f
    };
```

5.2.2.7 Inverse Quantization Table for the Exponential Window Smoothing Parameter Lambda

Non-uniform inverse quantization of parameter lambda in range from 0 to 0,9999 with higher resolution approaching 1.

```
float m_rExpWinLambdaTable[32] =
{
         0.0000f,   0.1535f,   0.3000f,   0.4335f,   0.5504f,   0.6491f,   0.7299f,   0.7944f,
         0.8449f,   0.8838f,   0.9134f,   0.9357f,   0.9524f,   0.9648f,   0.9741f,   0.9809f,
         0.9860f,   0.9897f,   0.9924f,   0.9944f,   0.9959f,   0.9970f,   0.9978f,   0.9984f,
         0.9988f,   0.9991f,   0.9993f,   0.9995f,   0.9996f,   0.9997f,   0.9998f,   0.9999f
};
```

5.2.3 Helper Functions

5.2.3.1 ExtractVarLenBitFields

This routine is used to extract a field of variable length. The extracted field is interpreted as an unsigned integer (uint). The length of a field depends on the prefix code that is transmitted prior to the field itself and the values in the 4 element table passed through the input argument *uctable[]*.This function is described in pseudo-code in Table 5-2.

The prefix codes are 0b, 10b, 110b and 111b correspond to the index unindex = 0, 1, 2, and 3 respectively.

The corresponding field length is *uctable[unindex]*.

bExtractandAddFlag - is an optional parameter indicating if the extracted parameter value shall be added to a prefix index based offset. The default value is TRUE.

The output parameter is interpreted as a uint.

Table 5-2: ExtractVarLenBitFields

Syntax

```
// static tables
uint CBitStream::m_unIndexTbl[8] = {0,0,0,0,1,1,2,3};
uint CBitStream::m_unBitsUsed[8] = {1,1,1,1,2,2,3,3};

uint CBitStream::ExtractVarLenBitFields(uchar uctable[], bool bExtractandAddFlag)
{
  uint uncode;
  uint unindex;
  uncode = ExtractBits(3);
  RewindBitReadPointer(3 - m_unBitsUsed[uncode]); // Rewind unused bits
  // Look-up the index corresponding to the prefix code.
  unindex = m_unIndexTbl[uncode];
  // Calculate the parameter value:
  // - extract the field of length ucTable[unindex] and
  // - add the extracted value to the sum of (1<<uctable[un]) for all 0 <= un < unindex
  // if bExtractandAddFlag is TRUE, add the extraced value
  uint unvalue=0;
  if (uctable[unindex]>0) {
    if (bExtractandAddFlag) {
      for (uint un = 0; un<unindex; un++) {
        unvalue += (1 << uctable[un]);
      }
      unvalue = unvalue + (uint) ExtractBits( uctable[unindex] );
    }
    else {
      unvalue = (uint)ExtractBits(uctable[unindex]);
  }
  return unvalue;
}

void RewindBitReadPointer(uint unNumBits){ m_unBitCounter -= unNumBits; }
```

5.2.3.2 UpdateCode

Each parameter has its presence flag (*m_bParamPresent*) that is transmitted only if *bSyncOrFirstFrameFlag* is TRUE. If *m_bParamPresent* is FALSE, then no other fields related to this parameter are transmitted in any other frame. Instead, this parameter is by default set to either:

- *m_tSyncFramePredefValue* if the *m_bSyncFramePredefValueExists* is TRUE; or
- 0 if the *m_bSyncFramePredefValueExists* is FALSE.

where *m_bSyncFramePredefValueExists* is set during the instantiation of the parameter class.

If *m_bParamPresent* is TRUE and if *bFirstSamplPointofFirstFrame* is TRUE, then *m_bUpdateFlag* is:

- extracted from the stream if *m_bSyncFramePredefValueExists* is TRUE; or

- by default set to TRUE if *m_bSyncFramePredefValueExists* is FALSE.

If *m_bParamPresent* is TRUE and if *bFirstSamplPointofFirstFrame* is FALSE, then *m_bUpdateFlag* is extracted from the stream.

If *m_bUpdateFlag* is TRUE, *parameter code* is extracted from the stream.

This function is described in pseudo-code in Table 5-3.

Table 5-3: UpdateCode

Syntax

```
Function for Updating Predefined Bit-width Parameter

// Function for checking if parameter needs updating and if it does
// it performs the update of the parameter value.
// This is applicable to predefined bit-width parameters.
//
//  - bSyncOrFirstFrameFlag flag indicating if this is the sync frame, or the frame
//    between the two consecutive frames when a corresponding object appears for the first time.
//  - nBits the number of bits used for coding the parameter
// Output parameters:
//  - m_bUpdateFlag flag indicating if parameter has been updated or not
uint UpdateCode(bool bSyncOrFirstFrameFlag, uint nBits)
{
  uint unParameterCode;
  if (bSyncOrFirstFrameFlag) {
    // Extract a flag  m_bParamPresent that indicates the presence of the parameter in the stream
    m_bParamPresent =  (ExtractBits(1)==1) ? TRUE : FALSE;
    if (m_bParamPresent) {
      m_bUpdateFlag = (m_bSyncFramePredefValueExists) ? (bool) ExtractBits(1) : TRUE;
    }
    else{
      m_bUpdateFlag = FALSE;
    }
    if (m_bUpdateFlag) {
      // If m_bUpdateFlag is TRUE extract nBits for the new value of unParameterCode.
      unParameterCode =  (uint) ExtractBits(nBits);
    } else {
      unParameterCode = (uint) 0;
      m_tParameter = (m_bSyncFramePredefValueExists) ? m_tSyncFramePredefValue : (T) 0;
    }
  }
  else {
    m_bUpdateFlag = (m_bParamPresent) ? (bool) ExtractBits(1) : FALSE;
    // If m_bUpdateFlag is TRUE extract nBits for the new value of unParameterCode.
    // Otherwise return 0.
    unParameterCode = (m_bUpdateFlag) ? (uint) ExtractBits(nBits) : 0;
  }
  return unParameterCode;
}

// This version is applicable to variable bit-width parameters.
// Input Parameters:
//  - bSyncOrFirstFrameFlag flag indicating if this is the sync frame or the frame
//    between the two consecutive frames when a corresponding object appears for the first time
//  - uctable[4] table defines the length of a code word depending on extracted prefix code index
//
// Output parameters:
//  - m_bUpdateFlag flag indicating if parameter has been updated or not
uint UpdateCode(bool bSyncOrFirstFrameFlag, uint uctable[])
{
  // The parameter can get updated in any frame.
  if (bSyncOrFirstFrameFlag) {
    // Extract a flag  m_bParamPresent that indicates the presence of the parameter in the stream
    m_bParamPresent = (bool) ExtractBits(1);
    if (m_bParamPresent) {
      m_bUpdateFlag = (m_bSyncFramePredefValueExists) ? (bool) ExtractBits(1) : TRUE;
    } else {
      m_bUpdateFlag = FALSE;
    }
    if (m_bUpdateFlag) {
      unParameterCode =  (uint) ExtractVarLenBitFields(uctable);
    }
    else {
      unParameterCode = (uint) 0;
```

Syntax

m_tParameter = (m_bSyncFramePredefValueExists) ? m_tSyncFramePredefValue : (T) 0;
}
}
else {
m_bUpdateFlag = (m_bParamPresent) ? (bool) ExtractBits(1) : FALSE;
unParameterCode = (m_bUpdateFlag) ? (uint) ExtractVarLenBitFields(uctable) : 0;
}
return unParameterCode;
}

5.2.3.3 CountBitsSet_to_1

This function calculates the number of bits set to 1 in an unsigned 32-bit mask. This function is described in Table 5-4.

Table 5-4: CountBitsSet_to_1

Syntax

uint CBitStream::CountBitsSet_to_1(uint uintMask)
{
uint nb=0;
for (uint n=0; n<32; n++)
{
nb = ((uintMask>>n) & 1) ? (nb+1) : nb;
}
return nb;
}

5.2.3.4 GetPtrToCurrentByte

Function to get pointer to the current byte in the stream buffer. This function is described in Table 5-5.

Table 5-5: GetPtrToCurrentByte

Syntax

uchar * GetPtrToCurrentByte(void)
{
return m_ucStreamBuffer.data() + (m_unBitCounter >> 3);
}

5.2.3.5 GetBitCounter

Return the current bit counter. This function is described in Table 5-6.

Table 5-6: GetBitCounter

Syntax

uint GetBitCounter(void){
return m_unBitCounter;
}

5.2.3.6 AdvanceBitReadPointer

This function advances the bitstream read pointer, allowing the parser to skip some number of bits. This function is described in Table 5-7.

Table 5-7: AdvanceBitReadPointer

Syntax

void AdvanceBitReadPointer(uint unNumBits)
{
m_unBitCounter += unNumBits;
}

5.3 Interfaces for Extracting Metadata Frames

5.3.1 Overview of APIs

APIs for unpacking frames of metadata depend on the type of audio presentation selection. These parameters are dependent on the listening environment:

- *unOutChMask* is from the player API call set according to the user's selected playback configuration;
- *unMaxObjects* is a decoder configuration setting based on that particular decoder's capabilities (or resource allocation).

5.3.2 Audio Presentation Selection

The class for CAuPresSelectAPI is described in Table 5-8.

Table 5-8: CAuPresSelectAPI

Syntax

```
class CAuPresSelectAPI
{
private:

public:
  // API parameters
  uchar m_ucAudPresInterfaceType; // Type of API that was used to select desired audio presentation
  bool m_bEnblDefaultAuPres;
  uchar m_ucDesiredAuPresIndex;
  std::vector<uint> m_unDesiredObjIDList;
  std::vector<bool> m_bMatchFoundDesiredObjIDList;
  std::vector<uint> m_unMatchedAuPresIndex;
  std::vector<uint> m_unMatchedObjIndex;

  CAuPresSelectAPI(void):
    m_ucAudPresInterfaceType(AUD_PRES_SELECT_DEFAULT_AP),
    m_bEnblDefaultAuPres(FALSE),
    m_ucDesiredAuPresIndex(0),
    m_unDesiredObjIDList(),
    m_bMatchFoundDesiredObjIDList(),
    m_unMatchedAuPresIndex(),
    m_unMatchedObjIndex()
  {}

  ~CAuPresSelectAPI(void){}
  void CopyInputClassParams(CAuPresSelectAPI *pC)
  {
    pC->m_ucAudPresInterfaceType = m_ucAudPresInterfaceType;
    pC->m_bEnblDefaultAuPres = m_bEnblDefaultAuPres;
    pC->m_ucDesiredAuPresIndex = m_ucDesiredAuPresIndex;
    pC->m_unDesiredObjIDList = m_unDesiredObjIDList;
    // Note that the m_bMatchFoundDesiredObjIDList is not copied
  }
}
```

5.3.3 Play Default Presentation

Enable or disable the default audio presentation; the default objects within that presentation will be played or outputs will be muted respectively. Pseudo-code for this call is shown in Table 5-9.

Table 5-9: Default Presentation

Syntax	Reference

void CMetaDataFrame::UnpackMDFrame_API(bool bEnblDefaultAuPres,
uint unOutChMask,
uint unMaxObjects)
{
m_AuPrSelectAPI.m_ucAudPresInterfaceType = API_PRES_SELECT_DEFAULT_AP;
m_AuPrSelectAPI.m_bEnblDefaultAuPres = bEnblDefaultAuPres;
UnpackMDFrame(unOutChMask, unMaxObjects);
}
	6.2

5.3.4 Play Selected Presentation

Select a desired audio presentation other than the default presentation, and the default objects within that presentation will be played. Pseudo-code for this call is shown in Table 5-10.

Table 5-10: Selected Presentation

Syntax	Reference

void CMetaDataFrame::UnpackMDFrame_API(uchar ucDesiredAuPresIndex,
uint unOutChMask, uint unMaxObjects)
{
m_AuPrSelectAPI.m_ucAudPresInterfaceType = API_PRES_SELECT_SPECIFIC_AP;
m_AuPrSelectAPI.m_ucDesiredAuPresIndex = ucDesiredAuPresIndex;
UnpackMDFrame(unOutChMask, unMaxObjects);
}
	6.2

5.3.5 Play from a List of Objects

Play from a specified list of objects. In this case *unDepAuPresMask* indicates that all presentations are active, so the decoder shall determine which audio presentations carry objects from *m_AuPrSelectAPI_SampledatSyncFrm.m_unDesiredObjIDList*. Pseudo-code for this call is shown in Table 5-11.

Check whether any of the objects from *m_AuPrSelectAPI_SampledatSyncFrm.m_unDesiredObjIDList* is in the extracted object list *m_punObjectIDList[]*. Results are stored in the same memory locations as the explicit object list for this presentation that is extracted within FTOC.

In particular, the *ucAudPresIndex* bit within *m_pCFTOC->m_unDepAuPresExplObjListMask* is set to 1 if any explicitly requested objects are found within this audio presentation. Otherwise it is set to 0.

Table 5-11: Create Presentation from a List of Objects

Syntax	Reference

void CMetaDataFrame::UnpackMDFrame_API(std::vector<uint> unDesiredObjIDList,
uint unOutChMask, uint unMaxObjects)
{
m_AuPrSelectAPI.m_ucAudPresInterfaceType = API_PRES_SELECT_OBJECT_ID_LIST;
m_AuPrSelectAPI.m_unDesiredObjIDList = unDesiredObjIDList;
UnpackMDFrame(unOutChMask, unMaxObjects);
}
	6.2

6 DTS-UHD Metadata Header Parsing

6.1 Overview

This clause describes the metadata header, including the frame table of contents (FTOC).

6.2 UnpackMDFrame

This is the function for unpacking a frame of metadata where desired objects to be played are selected based on the type of API interface used (as indicated by *m_ucAudPresInterfaceType*) and the associated parameter for particular interface type. The class for CMetaDataFrame is shown in Table 6-1. Pseudo-code describing the extraction of the metadata frame is shown in Table 6-2.

Table 6-1: Frame Metadata Parameters

Syntax

```
// Class holding the frame of metadata
class CMetaDataFrame
{
private:
  CBitStream *m_pBitStream; // Pointer to a bitstream class
  uchar m_ucMaxNumAuPres; // Maximum number of audio presentations
  CCrc16 m_Crc16; // Holds class for calculation of CRC

public:
  typedef struct
  {
    uchar   m_ucAudPresInterfaceType
    bool    m_bEnblDefaultAuPres
    uchar   m_ucDesiredAuPresIndex
    std::vector< uint >     m_unDesiredObjIDList
    std::vector< bool >     m_bMatchFoundDesiredObjIDList
    std::vector< uint >     m_unMatchedAuPresIndex
    std::vector< uint >     m_unMatchedObjIndex
  } CAuPresSelectAPI;
  CAuPresSelectAPI m_AuPrSelectAPI; // Class holding the audio presentation selection API parameters
  CAuPresSelectAPI m_AuPrSelectAPI_SampledatSyncFrm; // Class holding the audio presentation
                                                     // selection API parameters sampled at sync frame
  CFrmTblOfContent *m_pCFTOC;     // Pointer to a CFrmTblOfContent class
  C0x01MDChunkBody **m_ppCMD0x01; // Pointer to an array of pointers to a C0x01MDChunkBody class
  CMFDStaticMD *m_pCMFDStaticMD;  // Pointer to a class with static MFD metadata
  uint *m_punBitStartExportChunks; // Pointer to the array with start bit index, within the frame
                                   // payload, corresponding to the beginning of MD ID of
                                   // each MD chunk with ID 0x02, 0x03 or 0x04.
  std::vector<uint> m_punObjectIDList; // Pointer to the vector with extracted object ID list
                                       // in current metadata chunk
    // Member functions
    ...
}
```

Table 6-2: UnpackMDFrame

Syntax	Reference

```
void CMetaDataFrame::UnpackMDFrame(uint unOutChMask, uint unMaxObjects)
{
  bool bSyncOK;
  uchar ucMDChunkID;
  uchar ucAudPresIndex;
  uint unDepAuPresMask;
  uint unBitCountBeforeMDID;
  uchar ucActiveAuPresIndex = 0;
  uchar ucExportChunkIndex = 0;
  uint unExtractedBits;
  uint unBits2Skip;

   // Extract Frame Table Of Content (FTOC)                                        6.4.3
  bSyncOK = m_pCFTOC->ExtractFTOC(m_AuPrSelectAPI, &m_AuPrSelectAPI_SampledatSyncFrm,
&m_Crc16);

  if (m_pCFTOC->m_bSyncFrameFlag){
        // In Sync Frame Reset All Metadata Chunks                                6.3.1
    ResetMDChunks();
  }
  // Get the mask indicating which dependent presentations shall be extracted
  unDepAuPresMask = m_pCFTOC->GetDepAuPresMask();                                 6.4.2

  // Loop over all metadata chunks
  for (uint unmdc = 0; unmdc < m_pCFTOC->m_unNum_MD_Chunks; unmdc++)
  {
```

Syntax	Reference
`// Check metadata CRC (if present) before proceeding with extraction of metadata`	
`if (m_pCFTOC->m_pbMDChunkCRCFlags[unmdc])`	6.3.8
`{`	
`  // Get pointer to the current read byte within the bitstream frame payload`	
`  uchar *ucpStreamBuff = GetPtrToCurrentByte();`	5.2.3.4
`}`	
`// Get Bit Counter at the beginning of the chunk`	
`unBitCountBeforeMDID = GetBitCounter();`	6.3.2
`// Extract MD Chunk ID`	
`ucMDChunkID = (uchar)ExtractBits(8);`	
`switch (ucMDChunkID)`	
`{`	
`case 0x01:`	
`  // 0x01 Metadata Chunk Carries Audio Presentation Index`	
`  ucAudPresIndex = ExtractAudPresIndex();`	6.3.3
`  // if current audio presentation shall be extracted`	
`  if ((unDepAuPresMask >> ucAudPresIndex) & 0x01)`	
`  {`	
`    ExtractMDChunkObjIDList();`	6.3.4
`    if (m_AuPrSelectAPI_SampledatSyncFrm.m_ucAudPresInterfaceType == API_PRES_SELECT_OBJECT_ID_LIST)`	
`    {`	
`      UpdateExplObjListforAuPres(ucAudPresIndex);`	6.3.6
`      if ((((m_pCFTOC->m_unDepAuPresExplObjListPresMask) >> ucAudPresIndex) & 0x01) == 0)`	
`      {`	
`       // No desired objects were found in this chunk, fix the read pointer`	
`        unExtractedBits = (GetBitCounter() - unBitCountBeforeMDID);`	
`       // Skip over the rest of this MD chunk`	
`        unBits2Skip = (8 * m_pCFTOC->m_punMDChunkSizes[unmdc]) - unExtractedBits;`	
`        AdvanceBitReadPointer(unBits2Skip);`	
`        // Exit switch (ucMDChunkID)`	
`        break;`	
`      }`	
`    } // End of if (m_AuPrSelectAPI_SampledatSyncFrm.m_ucAudPresInterfaceType`	
`    if (m_ppCMD0x01[ucActiveAuPresIndex] == NULL)`	
`    {`	
`      m_ppCMD0x01[ucActiveAuPresIndex] = new C0x01MDChunkBody(m_pBitStream, unMaxObjects);`	
`    }`	
`    m_ppCMD0x01[ucActiveAuPresIndex]->SetAudPresIndex(ucAudPresIndex);`	7.2
`    m_ppCMD0x01[ucActiveAuPresIndex]->SetNumOfObjects(m_punObjectIDList.size());`	
`    m_ppCMD0x01[ucActiveAuPresIndex]-> Extract_Main_0x01_MDChunk (`	7.4
`      m_pCFTOC->m_bSyncFrameFlag,`	
`      m_pCFTOC->m_bFullChannelBasedMixFlag,`	
`      unBitCountBeforeMDID,`	
`      m_pCFTOC->m_punMDChunkSizes[unmdc],`	
`      m_pCFTOC->m_unFrameDuration,`	
`      unOutChMask,`	
`      m_pCFTOC->m_bInteractObjLimitsPresent,`	
`      m_pCMFDStaticMD,`	
`      m_pCFTOC->m_pbAudPresSelectableFlag[ucAudPresIndex],`	
`      m_punObjectIDList);`	
`    if (((m_pCFTOC->m_unDepAuPresExplObjListPresMask) >> ucAudPresIndex) & 0x01)`	
`    {`	
`   // There is an explicit object list that shall be followed. Only the objects in`	
`    // the list with corresponding bit set to "1" within the indexed mask will be`	
`   // set to active and all others will be set to inactive. Static objects whose`	
`   // metadata is not transmitted within this frame maintain their activity state.`	
`      m_ppCMD0x01[ucActiveAuPresIndex]->ApplyExplObjList2ObjActMask(m_pCFTOC->m_punExplObjListMasks[ucAudPresIndex], m_punObjectIDList);`	7.3.1
`    }`	
`  // If any active object groups are in the object list, unwind the hierarchy and`	7.3.2
`    // set the referenced object activity flags accordingly.`	
`    m_ppCMD0x01[ucActiveAuPresIndex]->ApplyGroupDefaultObjectActivityMask();`	
`    ucActiveAuPresIndex++;`	
`  }`	
`  else`	
`  {`	
`    // This is not the desired Audio Presentation`	
`    unExtractedBits = (GetBitCounter() - unBitCountBeforeMDID);`	

Syntax	Reference

// Skip over the rest of this MD chunk
unBits2Skip = (8 * m_pCFTOC->m_punMDChunkSizes[unmdc]) - unExtractedBits;
AdvanceBitReadPointer(unBits2Skip);
}
break;
 	
case 0x02:
case 0x03:
case 0x04:
 	
// Save the bit pointer (pointing to the beginning of MD Chunk ID) for each export
// metadata chunk. The decoder will allow extraction of this metadata in the
// post processing modules.
// The post-processing module will copy this metadata into its local memory before the
// arrival of the next frame payload that will overwrite the decoder's copy.
m_punBitStartExportChunks[ucExportChunkIndex++] = unBitCountBeforeMDID;
 	
// Calculate the number of extracted bits within this Metadata Chunk
unExtractedBits = (GetBitCounter() - unBitCountBeforeMDID);
// Skip over the rest of this MD chunk
unBits2Skip = (8 * m_pCFTOC->m_punMDChunkSizes[unmdc]) - unExtractedBits;
AdvanceBitReadPointer(unBits2Skip);
break;
default:
// Calculate the number of extracted bits within this Metadata Chunk
unExtractedBits = (GetBitCounter() - unBitCountBeforeMDID);
//
// Skip over to the next MD chunk
unBits2Skip = 8 * m_pCFTOC->m_punMDChunkSizes[unmdc] - unExtractedBits;
AdvanceBitReadPointer(unBits2Skip);
 	
} // End of switch (ucMDChunkID)
 	
} // End of loop over all metadata chunks
} // End of UnpackMDFrame()	

6.3 Functions and Tables Supporting UnpackMDFrame

6.3.1 ResetMDChunks

Reset extracted metadata chunks according to the pseudo-code in Table 6-3.

Table 6-3: ResetMDChunks

Syntax

void CMetaDataFrame::ResetMDChunks(void)
{
// Reset 0x01 Metadata Chunks
for (uint nn = 0; nn < m_ucMaxNumAuPres; nn++)
{
if (m_ppCMD0x01[nn] != NULL)
{
delete m_ppCMD0x01[nn];
m_ppCMD0x01[nn] = NULL;
}
}
}

6.3.2 Metadata Chunk Types

The *Chunk_ID* is used to identify the type of data stored in the chunk according to the Metadata Chunk Types shown in Table 6-4.

The *Chunk_ID* is an 8-bit unsigned integer.

Multiple instances of the same chunk type (same *Chunk_ID*) are possible:

- when each instance is associated with different audio presentations; and/or
- when description of a single audio presentation exceeds the capabilities of a single metadata chunk instance.

Metadata chunks may appear within the stream in an arbitrary order.

Table 6-4: Metadata Chunk Types

Chunk_ID	Chunk Type Description
0x00	NULL chunk
0x01	Lean Object Metadata Chunk: frame info, object groups and objects metadata (Channel-mask based or Mixing metadata based, where mixing metadata can be 3D-based or based on output bus mixing coefficients)
0x02	DTS Post-Processing 1 Metadata (decoder exports this entire chunk via the API without unpacking it)
0x03	DTS Post-Processing 2 Metadata (decoder exports this entire chunk via the API without unpacking it)
0X04	DTS Post-Processing 3 Metadata (decoder exports this entire chunk via the API without unpacking it)
0x05 - 0xFE	Reserved
0xFF	Reserved for chunk types with extended Chunk_ID

6.3.3 Audio Presentation Index

Audio presentation index is part of the 0x01 metadata chunk and indicates the audio presentation index that is associated with this metadata chunk. The index values can be up to 0, 4, 20 or 31 depending on the transmitted variable length code of 1, 4 or 7 bits. Pseudo-code to extract the audio presentation index is shown in Table 6-5.

Table 6-5: ExtractAudPresIndex

Syntax	Reference

`// Function to extract the Audio Presentation Index`
`// for Metadata chunk types that explicitly carry this index`
`uchar CMetaDataFrame::ExtractAudPresIndex(void)`
`{`
`  uchar uctable[4] = { 0, 2, 4, 4 };`
`  uchar ucAudPresIndex = (uint)ExtractVarLenBitFields(uctable);`
`  return ucAudPresIndex;`
`}`
	5.2.3.1

6.3.4 ExtractMDChunkObjIDList

Extract the number of objects defined within a metadata chunk and the list of object IDs whose metadata is defined in the current frame of the MD chunk. The list is stored in *vector m_punObjectIDList* where *m_punObjectIDList.size()* indicates the number of objects. Pseudo-code for *ExtractMDChunkObjIDList*() is shown in Table 6-6.

Note that this list only refers to the object IDs packed in the current frame. Consequently, static objects are included in this list only in the frames when their metadata is transmitted (i.e. in the frame where a static object makes a first appearance in the period between the two sync frames).

Table 6-6: ExtractMDChunkObjIDList

Syntax	Reference

`// Function to extract the number of objects defined within a metadata chunk`
`// and the list of object IDs whose metadata is defined in the current frame`
 	
`void CMetaDataFrame::ExtractMDChunkObjIDList(void)`
`{`
`    bool bFullChannelBasedMixFlag = m_pCFTOC->m_bFullChannelBasedMixFlag;`
`    uint unNumObjects;`
 	
`  // Extract the number of objects defined in the chunk`
`  if (bFullChannelBasedMixFlag) {`
`    unNumObjects = 1;`
`  } else {`
`    uchar uctable[4] = { 3, 4, 6, 8 };`
	6.4.6.1
	
	6.3.5.1

Syntax	Reference

`    unNumObjects = (uint)ExtractVarLenBitFields(uctable) + 1;`
`  } // End of case when the bFullChannelBasedMixFlag=FALSE`
`  // case for extraction of m_unNumObjects`
`    // Initialize object list`
`    m_punObjectIDList.clear();`
`    m_punObjectIDList.resize(unNumObjects, 0);`
`    if (bFullChannelBasedMixFlag == FALSE){`
`        for (uint unobj = 0; unobj < unNumObjects; unobj++)`
`        {`
`            // Selects 4 or 8 bits for extraction of object ID`
`            uchar ucNumBitsforObjID = (ExtractBits(1) == 1) ? 8 : 4;`
`            // Extract Object ID`
`            m_punObjectIDList[unobj] = (uint)ExtractBits(ucNumBitsforObjID);`
`        }`
`    }`
`    else{`
`        // For full channel-based mixes set the Object ID to 256`
`        m_punObjectIDList[0] = 256;`
`    }`
`}`
	6.3.5.2
	6.3.5.3

6.3.5 Parameters in ExtractMDChunkObjIDList

6.3.5.1 m_unNumObjects

Total number of objects and object group definitions defined in this metadata chunk. This parameter can change from frame to frame. When *bFullChannelBasedMixFlag* is TRUE, *m_unNumObjects*=1 by default. When *bFullChannelBasedMixFlag* is FALSE *m_unNumObjects* is packed as a variable length (4, 6, 9 or 11 bits) code. Depending on the code length the maximum value for *m_unNumObjects* can be 8, 24, 88 or 256.

6.3.5.2 unNumBitsforObjID

This 1-bit field is extracted from the stream, for each of *m_unNumObjects objects*, only if the *bFullChannelBasedMixFlag* is FALSE. The *unNumBitsforObjID* is either 4 or 8 corresponding to the number of bits that are used to represent an object ID.

6.3.5.3 m_unObjectID

This field is extracted from the stream, for each of *m_unNumObjects* objects, only if the *bFullChannelBasedMixFlag* is FALSE. The *m_unObjectID* represents the unique object ID that can range from 0 to 15 (*if unNumBitsforObjID* = 4) or from 0 to 255 (if *unNumBitsforObjID* = 8). For full channel-based mixes (*bFullChannelBasedMixFlag* is TRUE) the Object ID is by default equal to 256.

6.3.6 Update Object List

A function to create or update an explicitly requested object list for a particular audio presentation based on:

- the global object list, provided by the system layer (*m_AuPrSelectAPI_SampledatSyncFrm.m_unDesiredObjIDList*); and
- the list of extracted objects in this 0x01 MD chunk (*m_punObjectIDList*[])

is shown in Table 6-7.

NOTE: Since *m_AuPrSelectAPI_SampledatSyncFrm.m_unDesiredObjIDList* can change only in sync frames, the static objects from this list will be activated whenever the respective metadata appears in the stream. Nominally this is in the frame when the static object first appears, in the interval between two sync frames. In all subsequent frames when its metadata is not transmitted, the static object activity flag is preserved.

Table 6-7: UpdateExplObjListforAuPres

Syntax

```
void CMetaDataFrame::UpdateExplObjListforAuPres(uchar ucAudPresIndex)
{
  uint unNumObj;
  uint unNumDesObj;
  uint unObjID;
  uint *unpDesID;
  uint unObjMask;
  bool bMatchFlag;
  unNumObj = m_punObjectIDList.size();
  unNumDesObj = m_AuPrSelectAPI_SampledatSyncFrm.m_unDesiredObjIDList.size();

  for (uint uno_api = 0; uno_api < unNumDesObj; uno_api++) {
    m_AuPrSelectAPI_SampledatSyncFrm.m_unMatchedObjIndex[uno_api] = 256;
  }
  unObjMask = 0; // Start from no match

  // Loop over all objects defined in this metadata chunk
  for (uint uno_md = 0; uno_md < unNumObj; uno_md++){
    unObjID = m_punObjectIDList[uno_md];
    unpDesID = m_AuPrSelectAPI_SampledatSyncFrm.m_unDesiredObjIDList.data();
    for (uint uno_api = 0; uno_api < unNumDesObj; uno_api++){
      if ((*unpDesID++) == unObjID) {
        bMatchFlag = m_AuPrSelectAPI_SampledatSyncFrm.m_bMatchFoundDesiredObjIDList[uno_api];
        if (bMatchFlag == FALSE) {
          m_AuPrSelectAPI_SampledatSyncFrm.m_bMatchFoundDesiredObjIDList[uno_api] = TRUE;
          // Set the uno_md-th bit of the unObjMask to 1
          unObjMask = unObjMask | (1 << uno_md);
          // Update Info Parameters
          m_AuPrSelectAPI_SampledatSyncFrm.m_unMatchedAuPresIndex[uno_api] = ucAudPresIndex;
          m_AuPrSelectAPI_SampledatSyncFrm.m_unMatchedObjIndex[uno_api] = uno_md;
          break; // Exit for (uint uno_api = 0; uno_api < unNumDesObj; uno_api++) loop
        }
      } // End of if ((*unpDesID++) == unObjID)
    } // End of for (uno_api = 0; uno_api < unNumDesObj; uno_api++)
  } // End of for (uint uno_md = 0; uno_md < unNumObj; uno_md++)
  if (unObjMask == 0) { // If no objects were found
    m_pCFTOC->m_unDepAuPresExplObjListPresMask =
                            m_pCFTOC->m_unDepAuPresExplObjListPresMask & (~(1 << ucAudPresIndex));
  }
  else {
    m_pCFTOC->m_unDepAuPresExplObjListPresMask = m_pCFTOC->m_unDepAuPresExplObjListPresMask | (1 <<
ucAudPresIndex);
  }
  // Save the matched list of explicit objects for this presentation
  m_pCFTOC->m_punExplObjListMasks[ucAudPresIndex] = unObjMask;
}
```

Results are:

- If any explicitly requested object is found within this audio presentation *m_pCFTOC->m_unDepAuPresExplObjListPresMask* is set to 1, otherwise it is set to 0.
- If *(m_pCFTOC->m_unDepAuPresExplObjListPresMask) >> ucAudPresIndex) & 0x01)* is TRUE, then the updated explicit object list mask for the audio presentation indicated by *ucAudPresIndex.*
- If *(m_pCFTOC->m_unDepAuPresExplObjListPresMask) >> ucAudPresIndex) & 0x01)* is FALSE, then *m_pCFTOC->m_punExplObjListMasks[ucAudPresIndex]* is set to 0, and the remainder of this metadata chunk shall be discarded and reading shall continue from the next metadata chunk.

6.3.7 Reserved and Byte Alignment Fields

To allow for the addition of a new metadata field within this chunk, all decoders shall track the number of bits extracted within this metadata chunk, compare it against the metadata chunk size and navigate past any remaining bits of the present metadata chunk.

6.3.8 Metadata Chunk CRC

The metadata chunk CRC (16-bit) value is transmitted in order to verify the chunk data. This CRC value is calculated over metadata fields starting from and including the MD Chunk ID and up to and including the byte alignment field prior to the CRC word (*unMDChunkSize*-2 bytes in total).

The decoder shall:

- Calculate the CRC(16) value over the chunk data fields i.e. over *unMDChunkSize* - 2 bytes (excluding CRC itself) of data.
- Extract the chunk CRC field and compare it against the calculated CRC(16) value.
- If the two values match, reverse back and start extracting the chunk fields; otherwise, pronounce data corruption.

The above steps are demonstrated in Table 6-8.

Table 6-8: ExtractandCheckCRC

Syntax	Reference

`bool CCrc16::ExtractandCheckCRC16(uchar *ucpStreamBuff, uint unSizeInBytes)`
`{`
`  unsigned short unCalcCrc16 = Calculate_CRC16(ucpStreamBuff, unSizeInBytes - 2);`
`  unsigned short unExtractedCRC16 = ((ucpStreamBuff[unSizeInBytes - 1] << 8) |`
`  ucpStreamBuff[unSizeInBytes]);`
`  // Check if calculated and extracted CRC16 words match`
`  bool bCRCVerifiedOK = (unCalcCrc16 == unExtractedCRC16) ? TRUE : FALSE;`
`  return bCRCVerifiedOK;`
`}`
	Table 6-9

Table 6-9: CRC16_Update4BitsFast

Syntax

`unsigned short CCrc16::m_CRC16_Lookup[16] = {`
`  0x0000, 0x01021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,`
`  0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF`
`};`
 	
`// Process 4 bits of the message to update the CRC Value.`
`// Note that the data will be in the low nibble of val.`
`void CCrc16::CRC16_Update4BitsFast(unsigned char val) {`
`  unsigned char  t;`
 	
`  // Step one, extract the most significant 4 bits of the CRC register`
`  t = m_CRC16Register >> 12;`
`  // XOR in the message Data into the extracted bits`
`  t = t ^ val;`
`  // Shift the CRC Register left 4 bits`
`  m_CRC16Register = m_CRC16Register << 4;`
`  // Do the table look-ups and XOR the result into the CRC Tables`
`  m_CRC16Register = m_CRC16Register ^ m_CRC16_Lookup[t];`
`}`
`// Process one Message Byte to update the current CRC Value`
`void CCrc16::CRC16_Update(unsigned char val) {`
`  CRC16_Update4BitsFast(val >> 4);    // High nibble first`
`  CRC16_Update4BitsFast(val & 0x0F);  // Low nibble`
`}`
`unsigned short CCrc16::Calculate_CRC16(unsigned char *buf, int len) {`
`  register int counter;`
 	
`  // Initialize the CRC to 0xFFFF as per specification`
`  m_CRC16Register = 0xFFFF;`
`  for (counter = 0; counter <len; counter++) {`
`  CRC16_Update(*buf++);`
`  }`
`  return m_CRC16Register;`
`}`

6.4 Frame Table of Contents (FTOC)

6.4.1 Overview of the FTOC

DTS-UHD frames have a single Frame Table of Contents to navigate to the Metadata Chunks and Audio Chunks. Table 6-11 describes how to parse this structure. These parameters apply to the entire stream (i.e. all audio and metadata chunks and audio presentations) and are transmitted in the stream only in the sync frame (i.e. when *m_bSyncFrameFlag* = TRUE). In frames where *m_bSyncFrameFlag* = FALSE**,** only the parameters that can change in value from frame to frame are transmitted.

6.4.2 FTOC Class Parameters

The FTOC Class Parameters are shown in Table 6-10. Extracting the FTOC parameters from the bitstream is shown in Table 6-11.

Table 6-10: CFrmTblofContent

Syntax

```
class CFrmTblOfContent
{

private:
  CBitStream *m_pBitStream;                       // Pointer to the bitstream class
  uint m_unTotalFramePayloadinBytes;              // Frame payload in bytes
  // Stream parameters
  uint m_unBaseDuration;                          // The base duration in clock periods.
  uint m_unClockRateInHz;                        // Defines Clock Period / Clock rate
  CParamUpdSyncOrAnyFrame<uint64> m_TimeStamp;   // Time code expressed in audio samples.
  uint m_unAudioSamplRate;        // Defines audio sampling rate as a multiple of the clock rate
  bool m_ReservedStream;
  uint m_unDepAuPresMask; // Mask indicating which presentations desired presentation depends on
  CPBRSmoothBufferDesc m_CPBRBuffDesc;    // Class with Peak Bit Rate (PBR) Smoothing parameters
  uchar m_ucNumAudioPres;                // Number of audio presentations defined in this stream
  std::vector<uint> m_unDepAuPresExplObjListPresMask_Vector;  // for each selectable presentation

public:
  bool m_bSyncFrameFlag;               // Flag indicating if the current frame is the sync frame
  uint m_unFTOCPayloadinBytes; // FTOC payload size including the sync word and the CRC in Bytes
  bool m_bFullChannelBasedMixFlag;                // indicated complete channel-based mix
  uint m_unStreamMajorVerNum;                     //  Major stream version number
  uint m_unStreamMinorRevNum;                     //  Minor revision of the stream version number
  uint m_unFrameDuration;                         // The frame duration in clock periods
  bool m_bInteractObjLimitsPresent;               // indicates interactivity limit params

  // MD and Audio Chunk navigation parameters
  uint m_unNum_MD_Chunks;                         // Number of metadata chunks transmitted
  std::vector<bool> m_pbAudPresSelectableFlag;    // audio presentation selectable Flags
  std::vector<uint> m_punMDChunkSizes;            // MD chunk sizes;
  std::vector<bool> m_pbMDChunkCRCFlags;          // Pointer to the vector with MD CRC Flags
  uint m_unNum_Audio_Chunks;                      // The number of audio chunks present
  std::vector<uint> m_punExplObjListMasks;        // Vector with explicit object list masks;
  CAudioChunkListHandler m_CAudioChunkNaviLits;   // holds audio chunk navigation parameters
  uint m_unDepAuPresExplObjListPresMask;          // desired selectable presentation entry

    // Member Functions

      // Private parameter get functions
      bool GetSyncFrameFlag(void){return m_bSyncFrameFlag;}
      bool GetFramePayloadInBytes(void){return m_unTotalFramePayloadinBytes;}
      bool GetFullChBasedMixFlag(void){return m_bFullChannelBasedMixFlag;}
      bool GetInteractiveMusicFlag(void){return m_bInteractiveMusicStream;}

      // Get mask indicating audio presentations that this presentation depends on
      uint GetDepAuPresMask(void){return m_unDepAuPresMask;}
      ...
};
```

6.4.3 Extract FTOC

Table 6-11: ExtractFTOC

Syntax	Reference
bool CFrmTblOfContent::ExtractFTOC(CAuPresSelectAPI AuPresSelectAPI, CAuPresSelectAPI *pAuPresSelectAPI_SampledatSyncFrm, CCrc16 *pCCrc16)	
{	
uint unSyncWord;	
uint unBitCounter = GetBitCounter(); // Get initial Position	
uchar *ucpStreamBuff = m_pBitStream->GetPtrToCurrentByte();	
unSyncWord = ExtractBits(32);	6.4.4.1
switch (unSyncWord)	
{	
case 0x40411BF2:	
m_bSyncFrameFlag = TRUE;	
break;	
case 0x71C442E8:	
m_bSyncFrameFlag = FALSE;	6.4.4.2
break;	
default:	
return FALSE; // invalid sync word	
}	
{	
uchar uctable[4] = { 5, 8, 10, 12 };	6.4.4.3
m_unFTOCPayloadinBytes = (uint)ExtractVarLenBitFields(uctable) + 1;	
}	
// Extract the stream parameters and check the CRC	
ExtractStreamParams(ucpStreamBuff, pCCrc16);	6.4.5
if (m_bSyncFrameFlag)	
{	
AuPresSelectAPI.CopyInputClassParams(pAuPresSelectAPI_SampledatSyncFrm);	5.3.2
if (pAuPresSelectAPI_SampledatSyncFrm->m_ucAudPresInterfaceType == API_PRES_SELECT_OBJECT_ID_LIST)	
{	
uint unDesObj = pAuPresSelectAPI_SampledatSyncFrm->m_unDesiredObjIDList.size();	
// Reset the match lists and initialize to FALSE	
pAuPresSelectAPI_SampledatSyncFrm->m_bMatchFoundDesiredObjIDList.clear();	
pAuPresSelectAPI_SampledatSyncFrm->m_bMatchFoundDesiredObjIDList.resize(unDesObj, FALSE);	
pAuPresSelectAPI_SampledatSyncFrm->m_unMatchedAuPresIndex.resize(unDesObj, 256);	
pAuPresSelectAPI_SampledatSyncFrm->m_unMatchedObjIndex.resize(unDesObj, 256);	
}	
}	
// Audio Presentations Parameters	
ResolveAudPresParams(*pAuPresSelectAPI_SampledatSyncFrm);	6.4.7
uint unCumulChunkPayloadInBytes = ExtractChunkNaviData();	6.4.13
// Save the total frame payload	
m_unTotalFramePayloadinBytes = m_unFTOCPayloadinBytes + unCumulChunkPayloadInBytes;	
// Extract XLL Peak Bit Rate (PBR) Smoothing Buffer Parameters	
ExtractPBRSmoothParams();	6.4.15
// Skip Reserved, Byte Alignment and CRC Fields	
//	
SkipReservedandAlignFields(m_bSyncFrameFlag, unBitCounter, m_unFTOCPayloadinBytes);	6.4.17
return TRUE;	
} // End of CFrmTblOfContent::ExtractFTOC()	

6.4.4 Parameters for FTOC

6.4.4.1 unSyncWord

This field represents a 32-bit sync word that, when expressed in hexadecimal format, is either **0x40411BF2** for sync frames or **0x71C442E8** in non-sync frames.

6.4.4.2 bSyncFrameFlag

The boolean *m_bSyncFrameFlag* is set TRUE if the sync word indicates the present frame being decoded is a sync frame, otherwise it is set to FALSE.

Decoders shall only attempt to initialize on a sync frame. After acquiring synchronization every frame shall be checked for the sync word to determine if the frame is a sync frame or a non-sync frame.

6.4.4.3 unFTOCPayloadinBytes

The value of *unFTOCPayloadinBytes* indicates the size, in bytes, of data in FTOC which starts from the FTOC sync word through (and including) the FTOC CRC word (if present). The variable length coding allows for FTOC payload of 32, 288, 1 312 or 5 408 bytes.

6.4.5 Stream Parameters

The stream parameters are extracted as shown in Table 6-12.

Table 6-12: ExtractStreamParams

Syntax	Reference

`// Input Arguments:`
`// ucpStreamBuff, pointer to the first byte of FTOC data block`
`// pCCrc16, pointer to an instance of CRC16 class`
`//`
`void CFrmTblOfContent::ExtractStreamParams(uchar *ucpStreamBuff, CCrc16 *pCCrc16)`
`{`
`  if (m_bSyncFrameFlag) {`
`    m_bFullChannelBasedMixFlag = (ExtractBits(1) == 1) ? TRUE : FALSE;`
`  }`
`  if (m_bFullChannelBasedMixFlag == FALSE || m_bSyncFrameFlag == TRUE) {`
`    if (pCCrc16->ExtractandCheckCRC16(ucpStreamBuff, m_unFTOCPayloadinBytes) == FALSE)`
`    {`
`      cerr << "CRC failed";`
`      exit(-1); // Exit`
`    }`
`  }`
`  if (m_bSyncFrameFlag)`
`  {`
`    uint unTemp;`
`    uint64  un64Temp;`
`    uchar ucTemp;`
`    if (m_bFullChannelBasedMixFlag == FALSE)`
`    {`
`      bool bShortRevNum = (ExtractBits(1) == 1) ? TRUE : FALSE;`
`      uchar ucBits = (bShortRevNum) ? 6 : 12;`
`      unTemp = ExtractBits(ucBits);`
`      ucBits = ucBits >> 1;`
`      // Select ucBits MSBs from the 2*ucBits-bit word unTemp`
`      // make the minor version start from 2`
`      m_unStreamMajorVerNum = (unTemp >> ucBits) + 2;`
`      // Select ucBits LSBs from the 2*ucBits-bit word unTemp`
`      m_unStreamMinorRevNum = unTemp & ((1 << ucBits) - 1);`
`    }`
`    else`
`    {`
`      m_unStreamMajorVerNum = 2;`
`      m_unStreamMinorRevNum = 0;`
`    }`
`    unTemp = ExtractBits(2);`
`    m_unBaseDuration = (uint)TableLookUp(untemp, BaseDuration);`
`    unTemp = ExtractBits(3) + 1;`
`    m_unFrameDuration = m_unBaseDuration * ((uint)unTemp);`
`    unTemp = ExtractBits(2);`
`    switch (unTemp) {`
`      case 0:`
`        m_unClockRateInHz = 32000;`
`        break;`
`      case 1:`
`        m_unClockRateInHz = 44100;`
`        break;`
	6.4.6.1
	6.4.6.2
	6.4.6.3
	6.4.6.4
	6.4.6.5
	6.4.6.6

Syntax	Reference

case 2:
m_unClockRateInHz = 48000;
break;
case default: //this would be an error
break;
}
 	
// Time code expressed in audio samples coded as 36-bit number.
//
un64Temp = (uint64)m_TimeStamp.UpdateCode(m_bSyncFrameFlag, 32);
if (m_TimeStamp.m_bUpdateFlag)
{
// Since time code is coded as 36-bit value extract 4 more bits
unTemp = (uint)ExtractBits(4);
un64Temp = (un64Temp << 4) | ((uint64)unTemp);
m_TimeStamp.SetParameter(un64Temp);
}
	6.4.6.7

// Audio Sample Rate Multiplier
ucTemp = (uchar)ExtractBits(2);
ucTemp = 1 << ucTemp;
m_unAudioSamplRate = m_unClockRateInHz * (uint)ucTemp
	6.4.6.8

if (m_bFullChannelBasedMixFlag){
m_reservedFlag = FALSE;
m_bInteractObjLimitsPresent = FALSE;
}
else{
m_reservedFlag = (bool) ExtractBits(1);
m_bInteractObjLimitsPresent = (bool) ExtractBits(1);
}
	6.4.6.10
	6.4.6.11

} // End of if (m_bSyncFrameFlag) } // End of CFrmTblOfContent::ExtractStreamParams() function	

6.4.6 Parameters for ExtractStreamParams

6.4.6.1 Full Channel-based Mix Flag

If *m_bFullChannelBasedMixFlag* = TRUE, then the content represents a full channel mask-based audio presentation (i.e. stereo, 5.1, 11.1 mix without objects). This implies the following:

- the stream version.revision number is 2.0;
- the number of audio presentations is 1;
- the number of audio chunks is 1;
- the number of metadata chunks in the sync frame is 1;
- there are no metadata chunks in non-sync frames;
- some metadata parameters will assume default values and will not exist in the stream.

6.4.6.2 CRC Test

When *m_bFullChannelBasedMixFlag* = TRUE, the FTOC CRC16 word is transmitted in the stream only in the sync frames. Consequently for this case the CRC check is performed only in sync frames.

When *m_bFullChannelBasedMixFlag* = FALSE, the FTOC CRC16 word is transmitted in every frame, so the CRC check is performed in all frames.

6.4.6.3 Stream Version Number

The stream version number is in a decimal representation in form of x.y where x represents a major version number and y represents a minor revision of the x version.

A DTS-UHD decoder implementation, with its version as indicated by the REVISION_NUMBER, shall do the following:

- Decode all stream versions x.y if the major version number x <= REVISION_NUMBER.
- Reject all stream versions x.y if the major revision number x > REVISION_NUMBER.

Coding of the version and revision numbers is performed either using:

- 7-bit code to represent major versions from 2 to 9 and minor revisions from 0 to 7.
- 13-bit code to represent major versions from 2 to 65 and minor revisions from 0 to 63.

If *m_bFullChannelBasedMixFlag* = TRUE then no bits are extracted from the stream for Version Number and the version is set to 2.0.

6.4.6.4 Base Duration

m_unBaseDuration indicates the number of clock cycles in the base duration period. The 2-bit index is used to distinguish between 4 different base duration periods as indicated in Table 6-13.

Table 6-13: Base Duration

index	BaseDuration (samples)
0	512
1	480
2	384
3	Reserved

6.4.6.5 Frame Duration

m_unFrameDuration indicates the number of clock cycles (as defined by the *m_unClockRateInHertz*) in the frame duration period. It is derived from an extracted 3-bit multiplier and *m_unBaseDuration*.

6.4.6.6 Clock Rate

The clock rate is used to calculate the audio sampling rate. *m_unClockRateInHz* is a 2-bit field representing the clock rate in Hz according to Table 6-14.

Table 6-14: Clock Rate

code (from bitstream)	ClockRate (Hz)
0	32 000
1	44 100
2	48 000
3	Unused

6.4.6.7 Time Code Parameters

The *m_TimeStamp.m_bUpdateFlag* indicates if the time code parameter is transmitted in the stream. This flag is extracted/set within the *m_TimeStamp.UpdateCode()*. *UpdateCode()* is explained in clause 5.2.3.2.

The call to *m_TimeStamp.UpdateCode()* returns the status of *m_TimeStamp.m_bUpdateFlag*. If *m_TimeStamp.m_bUpdateFlag* = TRUE then an additional 4 bits shall be read from the stream to complete the timestamp update. The time stamp data is a 36-bit field composed as follows:

TimeStamp = (Hours x 3 600) + Mins x 60 + Sec) / (float) *m_unClockRateInHz* + *SampleOffset*

Where:

- Hours has range 0 to 23

- Mins has range 0 to 59
- Sec has range 0 to 59

The time stamp of an encoded frame (N) corresponds to that time when the first bit of the encoded frame (N) shall be clocked to the decoder (i.e. the decoder presentation time for frame (N)).

6.4.6.8 Audio Sampling Rate

m_unAudioSamplRateInHz represents the audio sampling rate in Hz. The 2-bit parameter read from the FTOC Parameters is used as a power of 2 (2^n) that takes results in a multiplier of 1, 2, 4 or 8. This is multiplied by the clock rate, *m_unClockRateInHz.*

6.4.6.9 Number of Audio Samples

The number of audio samples in the metadata frame duration period is calculated as *m_unFrameDuration* * *m_unAudioSamplRateInHz* / *m_unClockRateInHz.*

In the nominal case, the audio frame duration period in seconds and the metadata frame duration period in seconds are the same. However, there are exceptions to this nominal case. In particular when audio is packed in the audio chunk types that allow packing of multiple audio frames within a single audio chunk (i.e. Audio Chunk Types 0x03 or 0x04 as described in Audio Chunks), the audio frame duration period in seconds is a fraction of the metadata frame duration period in seconds. Consequently the number of audio samples in the audio frame can be calculated as:

m_unFrameDuration * (*m_unAudioSamplRateInHz* / *m_unClockRateInHz*) in the nominal case

m_unFrameDuration * (*m_unAudioSamplRateInHz* / *m_unClockRateInHz*) / 2 in the Audio Chunk Type 0x03 case or

m_unFrameDuration * (*m_unAudioSamplRateInHz* / *m_unClockRateInHz*) / 4 in the Audio Chunk Type 0x04 case

6.4.6.10 Reserved Flag

m_reservedFlag shall be read when *m_bFullChannelBasedMixFlag = FALSE.* If *m_bFullChannelBasedMixFlag =* TRUE then this parameter is not present.

6.4.6.11 Object Interactivity Limits Present Flag

m_bInteractObjLimitsPresent shall be read when *m_bFullChannelBasedMixFlag = FALSE.* If *m_bFullChannelBasedMixFlag* = TRUE then this parameter is not present. When *m_bInteractObjLimitsPresent* = TRUE, then the metadata for limiting the user control of interactive objects is present in the stream.

6.4.7 ResolveAudPresParams

Within every selectable audio presentation, a set of audio presentation parameters is transmitted in the stream. Each set consists of the parameters describing the default dependency of selected presentation on any audio presentations of the lower index. This is shown in Table 6-15.

Table 6-15: ResolveAudPresParams

Syntax	Reference

`// Function resolves whether the audio presentation parameters shall be extracted and saved`
`// or extracted and overwritten by desired presentation mask provided as an input argument.`
`// Arguments:`
`//  AuPresSelectAPI - class of Audio presentation selection API parameters`
`//`
`void CFrmTblOfContent::ResolveAudPresParams(CAuPresSelectAPI AuPrSelectAPIClass)`
`{`
`  uchar ucDesiredAuPresIndex;`
 	
`  if (m_bSyncFrameFlag) {`
`    if (m_bFullChannelBasedMixFlag){`
`      m_ucNumAudioPres = 1;`	

Syntax	Reference

`    }`
`    else {`
`      uchar uctable[4] = { 0, 2, 4, 5 };`
`      m_ucNumAudioPres = ((uint)ExtractVarLenBitFields(uctable)) + 1;`
`    }`
`  }`
	6.4.8.1

`  switch (AuPrSelectAPIClass.m_ucAudPresInterfaceType) {`
`  case    API_PRES_SELECT_DEFAULT_AP:`
`    ucDesiredAuPresIndex = 0;`
`    ExtractAudPresParams(ucDesiredAuPresIndex);`
`    break;`
	6.4.9

`  case    API_PRES_SELECT_SPECIFIC_AP:`
`    ucDesiredAuPresIndex = AuPrSelectAPIClass.m_ucDesiredAuPresIndex;`
`    ExtractAudPresParams(ucDesiredAuPresIndex);`
`    break;`
	6.4.9

`  case    API_PRES_SELECT_OBJECT_ID_LIST:`
`    uint unDesiredAuPresMask = (1 << m_ucNumAudioPres) - 1;`
`    ExtractandSaveAllAudPresParams();`
`    break;`
`  }`
`}`
	6.4.12.4

6.4.8 Parameter for ResolveAudPresParams

6.4.8.1 Number of Audio Presentations

The number of audio presentations defined in the stream is coded with 4 variable length codes (1, 4, 7 or 8 bit length), allowing for 1, 5, 21 or 32 audio presentations to be defined in total.

6.4.9 Extract Audio Presentation Parameters

ExtractAudPresParams is shown in Table 6-16.

Table 6-16: ExtractAudioPresParams

Syntax	Reference

`void CFrmTblOfContent::ExtractAudPresParams`
`(uchar ucDesiredSelectAuPresIndex,`
` bool bSaveParameters4AllPresentations)`
`{`
`  uint unDepAuPresMask;`
`  uchar ucSelectablePresCounter = 0;  //number of selectable audio presentations`
 	
`  if (m_bSyncFrameFlag)`
`  {`
`    // Reinitialize the audio presentation related vectors`
`    m_pbAudPresSelectableFlag.clear();`
`    m_pbAudPresSelectableFlag.resize(m_ucNumAudioPres, FALSE);`
`    m_punExplObjListMasks.clear();`
`    m_punExplObjListMasks.resize(m_ucNumAudioPres, 0);`
`    m_unDepAuPresExplObjListPresMask_Vector.clear();`
`    unDepAuPresExplObjListPresMask_Vector.resize(m_ucNumAudioPres, 0);`
`    // Initialize the masks to 0`
`    m_unDepAuPresMask = 0;`
`  }`
`  for (uchar ucAuPresInd = 0; ucAuPresInd < m_ucNumAudioPres; ucAuPresInd++)`
`  {`
`    // Index into the vectors of saved parameters`
`    unSavedIndex = (bSaveParameters4AllPresentations) ? ucAuPresInd : 0;`
`    if (m_bSyncFrameFlag) {`
`      if (m_bFullChannelBasedMixFlag) {`
`        m_pbAudPresSelectableFlag[ucAuPresInd] = TRUE;`
`      } else {`
`        m_pbAudPresSelectableFlag[ucAuPresInd] = (bool)ExtractBits(1);`
`      }`
`    } // End of case when the m_bSyncFrameFlag is TRUE`
`    if (m_pbAudPresSelectableFlag[ucAuPresInd] == TRUE) {`
`      uint unTempFlags;`
`      if (m_bSyncFrameFlag) {`
`        // Extract presentation dependency mask`
	6.4.10.1
	
	6.4.10.2

Syntax	Reference

unDepAuPresMask = (ucAuPresInd > 0) ? ExtractBits(ucAuPresInd) : 0;
unTempFlags = 0;
for (uint uniter = 0; uniter < ucAuPresInd; uniter++) {
if ((unDepAuPresMask >> uniter) & 0x01) {
// Flag indicating if an explicit object list is transmitted
uint unTemp = (uint)ExtractBits(1);
unTempFlags = unTempFlags | (unTemp << uniter);
}
}
m_ unDepAuPresExplObjListPresMask_Vector [ucAuPresInd] = unTempFlags;
} // End of case when the m_bSyncFrameFlag is TRUE
// Objects may appear and disappear so list may need updating in non-sync frames
unTempFlags = (m_bSyncFrameFlag) ? unTempFlags : m_unDepAuPresExplObjListMask;
// Lists are only saved if this is the desired presentation
bool bSaveListsFlag = (ucSelectablePresCounter == ucDesiredSelectAuPresIndex);
ExtractExplicitObjectLists(m_unDepAuPresExplObjListPresMask_Vector[ucAuPresInd], ucAuPresInd, bSaveListsFlag);
if (m_bSyncFrameFlag) {
if (bSaveParameters4AllPresentations || bSaveListsFlag)
{
// m_unDepAuPresExplObjListPresentMask
// will be overwritten within the ExtractandSaveAllAudPresParams()
// Since this is desired presentation save the m_unDepAuPresMask
// Include the ucAuPresInd
m_unDepAuPresMask = unDepAuPresMask | (1 << ((uint)ucAuPresInd));
// Save the mask associated with the desired selectable presentation
m_unDepAuPresExplObjListPresMask =
m_unDepAuPresExplObjListPresMask_Vector[ucAuPresInd];
} // End of case when the ucSelectablePresCounter = ucDesiredSelectAuPresIndex
} // End of case when the m_bSyncFrameFlag is TRUE
ucSelectablePresCounter++; // Increment selectable audio presentation counter
} // End of if (m_pbAudPresSelectableFlag[ucAuPresInd]==TRUE)
else
{
m_unDepAuPresExplObjListPresMask_Vector[ucAuPresInd] = 0;
}
} // End of audio presentation index loop
 	
if (m_bSyncFrameFlag && (bSaveParameters4AllPresentations==FALSE))
{
if ((m_ucNumAudioPres == 1) &&
(ucDesiredSelectAuPresIndex==0) &&
(m_pbAudPresSelectableFlag[0] == FALSE)) {
m_unDepAuPresMask = 1;
}
}
}
	6.4.10.3
	6.4.11

6.4.10 Parameters for ExtractAudPresParams

6.4.10.1 bAudPresSelectableFlag

This 1-bit field is transmitted in the stream only if the *m_bSyncFrameFlag* is TRUE. The presentation selectable flag indicates if an audio presentation with all its dependent lower indexed presentations (as indicated by *m_unDepAuPresMask*) is playable without any higher indexed presentations. In particular when:

- *bAudPresSelectableFlag* = FALSE this audio presentation cannot be selected as a desired presentation (top of the dependency chain). However, it can be played as a part of dependency chain of some presentation with a higher presentation index;
- if *bAudPresSelectableFlag* = TRUE this audio presentation can be selected as a desired presentation (top of the dependency chain) and consequently the dependency mask *unDepAuPresMask* is transmitted in the stream. The *unDepAuPresMask* indicates which of the lower-indexed audio presentations shall be by default played together with the desired audio presentation.

6.4.10.2 unDepAuPresMask

This field is transmitted only if the *m_bSyncFrameFlag* is TRUE and only for selectable presentations (i.e. *bAudPresSelectableFlag* = TRUE). The *ucDepAuPresMask* is a bit mask describing which of the lower indexed presentations shall be by default played together with the current selectable audio presentation. Since a selectable presentation with audio presentation index *ucAudPresIndex* can only depend on audio presentations with indices smaller than *ucAuPresInd*, then a *ucAuPresInd*-bit long dependency mask is needed. The dependency of the current audio presentation on the audio presentation with *ucAuPresInd=k* is captured by the *k*-th bit (*k*=0 for LSB) of *ucDepAuPresMask*. In particular if the *k*-th bit in the mask is "1" the current audio presentation depends on the audio presentation with presentation index *k*.

6.4.10.3 Explicit Object List Present Flag

This 1-bit field is transmitted only if the *m_bSyncFrameFlag* is TRUE and only for selectable presentations (i.e. *bAudPresSelectableFlag* = TRUE). For each dependent presentation, extract a bit that indicates if the default objects within that presentation shall be played (bit set to '0') or there exists an explicitly defined list of objects that are to be played instead (bit set to '1'). The flags corresponding to all dependent audio presentations are aggregated into a single mask and saved to the *m_unDepAuPresExplObjListPresMask_Vector[]*. The saved values in *m_unDepAuPresExplObjListPresMask_Vector[]* are needed for extraction of explicit object lists in the non-sync frames.

The value in *m_unDepAuPresExplObjListPresMask_Vector[]* that corresponds to the desired selectable presentation is saved in *m_unDepAuPresExplObjListPresMask*.

6.4.11 ExtractExplicitObjectsLists

The following fields are transmitted in the stream for each lower indexed presentation that has its corresponding bit within *unDepAuPresExplObjListMask* set to "1" according to the pseudo-code in Table 6-17.

Table 6-17: ExtractExplicitObjectsLists

Syntax	Reference

`void CFrmTblOfContent::ExtractExplicitObjectLists(`
`  uint unDepAuPresExplObjListMask,`
`  uchar uCurrentAuPresInd,`
`  bool bSaveListsFlag) {`
 	
`  uint unTemp;`
`  bool bUpdFlag;`
 	
`  for (uchar ucapi = 0; ucapi < uCurrentAuPresInd; ucapi++) {`
`  if ((unDepAuPresExplObjListMask >> ucapi) & 0x01) {`
`  // Check if there is an update of the list mask`
`  if (m_bSyncFrameFlag) {`
`  // In sync frames there will be an update`
`  bUpdFlag = TRUE;`
`  }`
`  else {`
`  bUpdFlag = (ExtractBits(1) == 1) ? TRUE : FALSE;`
`  }`
`  if (bUpdFlag) {`
`  uchar uctable[4] = { 4, 8, 16, 32 };`
`  unTemp = ((uint)ExtractVarLenBitFields(uctable));`
`  }`
`  } // End of if ((unDepAuPresExplObjListMask >> ucapi) & 0x01)`
`  else {`
`  unTemp = 0;`
`  bUpdFlag = TRUE;`
`  }`
`  m_punExplObjListMasks[ucapi] = (bSaveListsFlag && bUpdFlag) ? unTemp :`
`m_punExplObjListMasks[ucapi];`
`  }`
`}`
	6.4.12.1
	6.4.12.2

6.4.12 Parameters for ExtractExplicitObjectLists

6.4.12.1 Explicit Object List Update Flag

This 1-bit flag is transmitted in the stream only if the *m_bSyncFrameFlag* is FALSE. In the case where *m_bSyncFrameFlag* is TRUE, the explicit object list update flag is set to TRUE by default. When this update flag is TRUE it indicates that the explicit object list mask is transmitted in the current frame. Otherwise, the explicit object list mask keeps its value from the previous frame.

6.4.12.2 Explicit Object List Mask

This variable length field represents an object list mask where each bit corresponds to an object within specific lower indexed audio presentation in the order in which objects are packed in the stream. When the bit is set to "1" it indicates that the playback of the corresponding object is requested.

NOTE: This list only refers to the object IDs packed in the current frame. Consequently static objects are included in this list only in the frames when their metadata is transmitted (i.e. the frame a static object makes a first appearance in the period between the two sync frames).

The explicit object list may have up to 4, 8, 16 or 32 objects using variable length codes of 5, 10, 19 or 35 bits respectively.

6.4.12.3 Metadata Describing Presentation Scaling for Different Playback Configurations

This metadata block is present only in the sync frames (m_*bSyncFrameFlag* is TRUE) and only for the audio presentations with *bAudPresSelectableFlag* equal to TRUE. This metadata block defines audio presentation scaling tailored for different playback configurations as described in clause 7.5.1.

6.4.12.4 Extract and Save All Audio Presentation Parameters

ExtractAudPresParams() is a function to extract and save all audio presentation parameters, as shown in Table 6-18. The function is only executed when the stream decoder is configured with an object list. In this case there is no preferred playable presentation and the list of desired objects is provided. Since the partitioning of objects across the audio presentations is not known within FTOC, all presentations shall be considered, therefore all *m_ucNumAudioPres* bits are all set to "1" in *unDepAuPresMask*.

An overall long term loudness measurement set and scaling parameter set for every selectable audio presentation shall be saved. Once it is determined which audio presentations shall be played, the appropriate parameters will be selected.

All dependent presentation masks and explicit object lists transmitted within this stream are ignored.

Table 6-18: ExtractandSaveAllAudPresParams

Syntax	Reference

`//input arguments:`
`// - unDepAuPresMask  specifies which audio presentations within this stream shall be`
`//                    decoded`
 	
`void CFrmTblOfContent::ExtractandSaveAllAudPresParams(uint unDepAuPresMask)`
`{`
`  ExtractAudPresParams(0);`
`  if (m_bSyncFrameFlag)`
`  {`
`    // Set m_ucNumAudioPres LSBs to "1" indicating that all presentations are desired.`
`    m_unDepAuPresMask = (1 << m_ucNumAudioPres) - 1;`
`    // Make sure to ignore all object lists explicitly defined within the bitstream`
`    m_unDepAuPresExplObjListPresMask = 0;`
`    for (uint unAPInd = 0; unAPInd < m_ucNumAudioPres; unAPInd++)`
`    {`
`      m_unDepAuPresExplObjListPresMask_Vector[unAPInd] = 0;`
`      m_punExplObjListMasks[unAPInd] = 0;`
`    }`
`  }`
`}`
	6.4.9

6.4.13 Navigation Parameters for Metadata and Audio Chunks

The prototype for the CAudioChunkNaviParams class is shown in Table 6-19.

Table 6-19: Audio Chunk Navigation - Class Parameters

```
class CAudioChunkNaviParams
{
private:
public:
  uint m_unAudioChunkPackinIndex;         // Chunk's packing index within the current frame payload
  uint m_unAudioChunkID;                  // A registered audio chunk ID
  uint m_unAudioChunkSize;                // A registered audio chunk size in Bytes
  uint m_unAudioChunkIndex;               // Registered unique audio chunk indexes as used within MD
                                          // chunks for addressing audio waveforms.
  // For a registered audio chunk that is still present in the stream, the audio chunk index has a
  // constant value.
  bool m_bAudioChunkPresent;              // Flags indicating if a registered audio chunk is still
                                          // present in the current frame
// Member Function ...

};
```

The chunk navigation table is extracted according to the pseudo-code shown in Table 6-20.

Table 6-20: ExtractChunkNaviData

Syntax	Reference
`// Extraction of Metadata Chunks and Audio Chunks Navigation Parameters`	
`uint CFrmTblOfContent::ExtractChunkNaviData(void) {`	
`  bool bACIDsPresentFlag;`	
`  uint unCumulChunkPayloadInBytes = 0;`	
`  // The number of metadata chunks present in current frame`	
`  if (m_bFullChannelBasedMixFlag) {`	6.4.14.1
`    m_unNum_MD_Chunks = (m_bSyncFrameFlag) ? 1 : 0;`	
`  } else {`	
`    uchar uctable[4] = { 2, 4, 6, 8 };`	
`    m_unNum_MD_Chunks = (uint)ExtractVarLenBitFields(uctable);`	
`  }`	
`  if (m_punMDChunkSizes.size() < m_unNum_MD_Chunks) {`	
`    m_punMDChunkSizes.resize(m_unNum_MD_Chunks, 0);`	
`    m_pbMDChunkCRCFlags.resize(m_unNum_MD_Chunks, FALSE);`	
`  }`	
`  uchar uctable[4] = { 6, 9, 12, 15 };`	6.4.14.2
`  for (uint nmdc = 0; nmdc < m_unNum_MD_Chunks; nmdc++) {`	
`    // Size includes bits for MD Chunk ID`	
`    m_punMDChunkSizes[nmdc] = (uint)ExtractVarLenBitFields(uctable);`	
`    unCumulChunkPayloadInBytes += m_punMDChunkSizes[nmdc]; // accumulate chunk payload`	
`    //  MD chunk CRC flag`	
`    if (m_bFullChannelBasedMixFlag) {`	
`      // For full channel-based mix disable CRC in metadata chunk by default`	
`      m_pbMDChunkCRCFlags[nmdc] = FALSE;`	
`    } else {`	6.4.14.3
`      // Extract MD chunk CRC flag`	
`      m_pbMDChunkCRCFlags[nmdc] = (ExtractBits(1) == 1) ? TRUE : FALSE;`	
`    }`	
`  }`	
`  // The number of audio chunks present in the current frame`	
`  if (m_bFullChannelBasedMixFlag) {`	
`    m_unNum_Audio_Chunks = 1;`	
`  } else {`	6.4.14.4
`    uchar uctable[4] = { 2, 4, 6, 8 };`	
`    m_unNum_Audio_Chunks = (uint)ExtractVarLenBitFields(uctable);`	
`  }`	
`  // Audio Chunk Navigation Data`	
`  if (m_bSyncFrameFlag) {`	
`    m_CAudioChunkNaviLits.ResetList();`	6.4.14.5
`  } else {`	
`    m_CAudioChunkNaviLits.ResetAudioChunkPresentFlags();`	6.4.14.6
`  }`	
`  uint unAudioChunkIndex;`	
`  uint unAudioChunkID;`	
`  uint unAudioChunkSize;`	

Syntax	Reference
for (uint nac = 0; nac < m_unNum_Audio_Chunks; nac++) {	
// Audio chunk index	
//	
if (m_bFullChannelBasedMixFlag) {	
// In this case there can only be one audio chunk	
unAudioChunkIndex = 0;	
} else {	
uchar uctable[4] = { 2, 4, 6, 8 };	
unAudioChunkIndex = (uint)ExtractVarLenBitFields(uctable);	6.4.14.7
}	
uint unListIndex = m_CAudioChunkNaviLits.FindAudioChunkListIndex(unAudioChunkIndex, unAudioChunkSize, nac);	
if (m_bSyncFrameFlag){	
// In sync frame Audio Chunk IDs will be present	6.4.14.8
bACIDsPresentFlag = TRUE;	
}	
else{	
bACIDsPresentFlag = (m_bFullChannelBasedMixFlag) ? FALSE : (bool)ExtractBits(1);	6.4.14.9
}	
if (bACIDsPresentFlag) {	
uchar uctable[4] = { 2, 4, 6, 8 };	6.4.14.10
unAudioChunkID = (uint)ExtractVarLenBitFields(uctable);	
}	
// Store the Audio Chunk ID	
m_CAudioChunkNaviLits.m_pCAudioChunksNaviParams[unListIndex].m_unAudioChunkID = unAudioChunkID;	
}	
else {	
// Case when the bACIDsPresentFlag is FALSE	
if (m_CAudioChunkNaviLits.m_pCAudioChunksNaviParams[unListIndex].m_unAudioChunkID < 256)	
{	
// Get the audio chunk ID from the stored list	
unAudioChunkID = m_CAudioChunkNaviLits.m_pCAudioChunksNaviParams[unListIndex].m_unAudioChunkID;	
}	
}	
// Audio Chunk Size in Bytes	
uchar uctable[4] = { 9, 11, 13, 16 };	
unAudioChunkSize = (unAudioChunkID == 0) ? 0 : (uint)ExtractVarLenBitFields(uctable);	6.4.14.11
unCumulChunkPayloadInBytes += unAudioChunkSize; // Update cumulative chunk payload	
// Store audio chunk size	
m_CAudioChunkNaviLits.m_pCAudioChunksNaviParams[unListIndex].m_unAudioChunkSize = unAudioChunkSize;	
} // End of Audio Chunk Loop	
// Make available spaces in the list by resting the space occupied by any registered	
// audio chunks that did not appear in this frame	
m_CAudioChunkNaviLits.PurgeList();	
return unCumulChunkPayloadInBytes;	6.4.14.12
} // End of CFrmTblOfContent::ExtractChunkNaviData()	

6.4.14 Parameters for ExtractChunkNaviData

6.4.14.1 Number of Metadata Chunks

This field indicates the number of metadata chunks present in the current frame. In the case where *m_bFullChannelBasedMixFlag* is TRUE, the number of metadata chunks is not transmitted and takes the default value of 1 in sync frames and 0 in non-sync frames. In the case where *m_bFullChannelBasedMixFlag* is FALSE, 4 variable length codes (3, 6, 9 or 11 bits in length) are used to allow encoding of up to 3, 19, 83 or 255 metadata chunks respectively.

6.4.14.2 Sizes of Metadata Chunks

These fields indicate the size in bytes of each metadata chunk present in the current frame. Four variable length codes (7, 11, 15 or 18 bits in length) are used to allow sizes of up to 63, 575, 4 671 or 32 767 bytes respectively.

6.4.14.3 Metadata Chunks CRC Flag

Each metadata chunk has a corresponding 1-bit flag indicating the presence of 16-bit CRC at the end of metadata chunk payload. The m*_pbMDChunkCRCFlags[nmdc]* stores the metadata CRC flag for metadata chunk with index nmdc. The flag is not transmitted if *m_bFullChannelBasedMixFlag* is TRUE and instead *m_pbMDChunkCRCFlags[nmdc]* is by default set to FALSE (indicating no CRC word in metadata chunk nmdc).

6.4.14.4 Number of Audio Chunks

This field indicates the number of audio chunks present in the current frame. When *m_bFullChannelBasedMixFlag* is TRUE, the number of audio chunks is not transmitted and takes the default value of 1. When *m_bFullChannelBasedMixFlag* is FALSE, 4 variable length codes (3, 6, 9 or 11 bits in length) are used to allow encoding of up to 3, 19, 83 or 255 audio chunks respectively.

6.4.14.5 Audio Chunk Handler ResetList

Resets all list members but keeps the list size, as demonstrated in Table 6-21.

Table 6-21: ResetList

Syntax

```
void CAudioChunkListHandler::ResetList(void)
{
  for (uint unli = 0; unli < m_unHigestIndex; unli++)
  {
    m_pCAudioChunksNaviParams[unli].m_bAudioChunkPresent = FALSE;
    m_pCAudioChunksNaviParams[unli].m_unAudioChunkID = 0;
    m_pCAudioChunksNaviParams[unli].m_unAudioChunkIndex = 0;
    m_pCAudioChunksNaviParams[unli].m_unAudioChunkPackinIndex = 0;
    m_pCAudioChunksNaviParams[unli].m_unAudioChunkSize = 0;
  }
  m_unHigestIndex = 0;
} // End ResetListMembers();
```

6.4.14.6 ResetAudioChunkPresentFlags

Resets *m_pCAudioChunksNaviParams[].m_bAudioChunkPresent* as shown in Table 6-22.

Table 6-22: ResetAudioChunkPresentFlags

Syntax

```
void CAudioChunkListHandler::ResetAudioChunkPresentFlags(void)
{
  for (uint unli = 0; unli < m_unHigestIndex; unli++) {
    m_pCAudioChunksNaviParams[unli].m_bAudioChunkPresent = FALSE;
  }
} // End ResetAudioChunkPresentFlags();
```

6.4.14.7 Audio Chunk Index

This field indicates an audio chunk index, which is unique for each audio chunk. This index is used for referencing audio chunks within the metadata chunks. This index has a constant value over the life time of an audio chunk. Four variable length codes (3, 6, 9 or 11 bits in length) are used to allow chunk index values up to 3, 19, 83 or 255 respectively.

6.4.14.8 FindAudioChunkListIndex

Function that finds an index in the list of an audio chunk with given audio chunk index, as demonstrated in Table 6-23. If the *unDesiredAudioChunkIndex* is not in the list, the first available space in the list is returned. For every matched audio chunk index the corresponding *m_bAudioChunkPresent* is set to TRUE.

Table 6-23: FindAudioChunkListIndex

Syntax

```
uint CAudioChunkListHandler::FindAudioChunkListIndex(uint unDesiredAudioChunkIndex, uint
unCorrespondingPackIndex)
{
  uint unFirstAvailablePlace = m_unHigestIndex; // Assume no available spaces in the list

  for (uint unli = 0; unli < m_unHigestIndex; unli++) {
    if (m_pCAudioChunksNaviParams[unli].m_unAudioChunkIndex == unDesiredAudioChunkIndex) {
      // Found match in the list return the list index
      m_pCAudioChunksNaviParams[unli].m_bAudioChunkPresent = TRUE;
      m_pCAudioChunksNaviParams[unli].m_unAudioChunkPackinIndex = unCorrespondingPackIndex;
      return unli;
    }
    // Check if this is an available place in the list
    if ((m_pCAudioChunksNaviParams[unli].m_bAudioChunkPresent == FALSE)
        && (m_pCAudioChunksNaviParams[unli].m_unAudioChunkSize == 0)) {
      unFirstAvailablePlace = (unFirstAvailablePlace<unli) ? unFirstAvailablePlace : unli;
    }
  }
  // No match has been found;
  uint unNewACListIndex;
  if (unFirstAvailablePlace < m_unHigestIndex) {
    // There has been an empty space so store the new audio chunk parameters
    // in position indicated by the unFirstAvailablePlace
    unNewACListIndex = unFirstAvailablePlace;
  } else {
    // Increase the list size by 1 and store the new AC in that position
    unNewACListIndex = m_unHigestIndex;
    m_unHigestIndex = m_unHigestIndex + 1;
  }
  m_pCAudioChunksNaviParams[unNewACListIndex].m_bAudioChunkPresent = TRUE;
  m_pCAudioChunksNaviParams[unNewACListIndex].m_unAudioChunkIndex = unDesiredAudioChunkIndex;
  m_pCAudioChunksNaviParams[unNewACListIndex].m_unAudioChunkPackinIndex = unCorrespondingPackIndex;
  // Audio Chunk ID is unknown so set to 256 indicating a new audio chunk in the list
  m_pCAudioChunksNaviParams[unNewACListIndex].m_unAudioChunkID = 256;
  return unNewACListIndex;

} // End of FindAudioChunkListIndex()
```

6.4.14.9 Audio Chunk ID Present Flag

This 1-bit flag when TRUE, indicates that the audio chunk ID is transmitted in the current frame. In sync frames this flag is not transmitted and takes the default value *bACIDsPresentFlag* is TRUE. In non-sync frames the flag shall not be transmitted if the *m_bFullChannelBasedMixFlag* is TRUE and takes the default value *bACIDsPresentFlag* is FALSE.

6.4.14.10 Audio Chunk ID

This field is transmitted only if the *bACIDsPresentFlag* is TRUE and it indicates an audio chunk ID. If the *bACIDsPresentFlag* is FALSE, the ID is inherited from the previously received ID. Four variable length codes (3, 6, 9 or 11 bits in length) are used to allow chunk ID values up to 3, 19, 83 or 255 respectively. The list of audio chunk types corresponding to different IDs is given in the Table of Audio Chunk Types.

6.4.14.11 Audio Chunk Size in Bytes

This field indicates the audio chunk's size (in Bytes) in the current frame. Four variable length codes (10, 13, 16 or 19 bits in length) are used to allow sizes of up to 511, 2 559, 10 751 or 65 535 bytes respectively.

6.4.14.12 Purge List

Indicates the position of previous audio chunks that have disappeared from the list are now available for new audio chunks, as demonstrated in Table 6-24. Availability is indicated by having:

m_pCAudioChunksNaviParams[unli].m_bAudioChunkPresent = FALSE; and

m_pCAudioChunksNaviParams[unli].m_unAudioChunkSize = 0

Table 6-24: CAudioChunkListHandler::PurgeList

Syntax

void CAudioChunkListHandler::PurgeList(void)
{
  for (uint unli = 0; unli < m_unHigestIndex; unli++) {
  if (m_pCAudioChunksNaviParams[unli].m_bAudioChunkPresent == FALSE) {
  m_pCAudioChunksNaviParams[unli].m_unAudioChunkSize = 0;
  }
  }
} // End PurgeList();

6.4.15 Peak Bit Rate Smoothing Parameters

The class prototype for CPBRSmoothBufferDesc is shown in Table 6-25. Pseudo-code to extract the smoothing buffer parameters is shown in Table 6-26.

Table 6-25: PBR Smoothing Buffer Class Parameters

Syntax

class CPBRSmoothBufferDesc
{
  // Class with Peak Bit Rate (PBR) Smoothing parameters
public:
  bool m_bVBRSmoothingBufferEnabled; // Flag indicating if PBR smoothing has been performed
  uint m_nuAudioChunksCumulSmoothPayload; // Size in bytes of smoothed payload
  bool m_bACFirstXLLSyncPresent; // Flag indicating if XLL sync word in frame smoothed payload
  uint m_nuVBRSmoothingBuffSzkB; // Size in bytes of required smoothing buffer
  uint m_nuInitACXLLDecDlyFrames; // Number of frames to buffer to start decoding frame payload
  uint m_nuACFirstXLLSyncOffsInDWORDS; // XLL Sync word offset (measured in 32-bit words)
  // Member functions
  ...
}

Table 6-26: Extract Peak Bit Rate (PBR) Smoothing Buffer Parameters

Syntax	Reference
void CFrmTblOfContent::ExtractPBRSmoothParams(void) {	
if (m_bSyncFrameFlag) {	
m_CPBRBuffDesc.m_bVBRSmoothingBufferEnabled = (ExtractBits(1)==1) ? TRUE : FALSE;	6.4.16.1
}	
if (m_CPBRBuffDesc.m_bVBRSmoothingBufferEnabled){	
uint nuBitsforCumACPayload = 9 + 3*ExtractBits(2);	6.4.16.2
m_CPBRBuffDesc.m_nuAudioChunksCumulSmoothPayload = ExtractBits(nuBitsforCumACPayload)+1;	6.4.16.3
m_CPBRBuffDesc.m_bACFirstXLLSyncPresent = ExtractBits(1);	6.4.16.4
if (m_CPBRBuffDesc.m_bACFirstXLLSyncPresent){	
m_CPBRBuffDesc.m_nuVBRSmoothingBuffSzkB = (ExtractBits(2)+1)<<6;	6.4.16.5
m_CPBRBuffDesc.m_nuInitACXLLDecDlyFrames=ExtractBits(8);	6.4.16.6
m_CPBRBuffDesc.m_nuACFirstXLLSyncOffsInDWORDS = ExtractBits(nuBitsforCumACPayload-2);	6.4.16.7
} // End of if (m_bACFirstXLLSyncPresent){	
} // End of if (m_bVBRSmoothingBufferEnabled){	
} // End of CFrmTblOfContent::ExtractVBRSmoothParams()	

6.4.16 Parameters for ExtractPBRSmoothParams

6.4.16.1 m_bVBRSmoothingBufferEnabled

m_bVBRSmoothingBufferEnabled is present in the stream only if *bSyncFrameFlag* = TRUE. In all other frames the value of *m_bVBRSmoothingBufferEnabled* is inherited from the previous sync frame. When *m_bVBRSmoothingBufferEnabled* = TRUE, the buffer for smoothing the variability in the cumulative payload of all audio chunks is enabled. In this case, the audio chunks payload is scheduled for transmission ahead of time and the smoothing buffer in the decoder is used to hold the payload until its decode time. The setup of this buffer is described by the parameters listed in the clauses that follow. If *m_bVBRSmoothingBufferEnabled* = FALSE, the smoothing buffer is not used (i.e. the buffer size is equal to 0).

6.4.16.2 nuBitsforCumACPayload

The *nuBitsforCumACPayload* parameter is present in the stream if *m_bVBRSmoothingBufferEnabled* = TRUE. *nuBitsforCumACPayload* indicates the number of bits used to represent *m_nuAudioChunksCumulSmoothPayload* described in clause 6.4.16.3 and *m_nuACFirstXLLSyncOffsInDWORDS*, described in clause 6.4.16.7.

Valid values for *nuBitsforCumACPayload* are 9, 12, 15 and 18.

6.4.16.3 m_nuAudioChunksCumulSmoothPayload

m_nuAudioChunksCumulSmoothPayload is present in the stream when *m_bVBRSmoothingBufferEnabled* = TRUE. This field indicates the size, in bytes, of the smoothed payload corresponding to all audio chunks that are transmitted in the current frame.

6.4.16.4 m_bACFirstXLLSyncPresent

m_bACFirstXLLSyncPresent is present in the stream if *m_bVBRSmoothingBufferEnabled* = TRUE. If *m_bACFirstXLLSyncPresent* = TRUE, the sync word of the first XLL frame header is present in the current frame payload. This indicates to the decoder that it should attempt to establish/verify synchronization in the current frame. If *m_bACFirstXLLSyncPresent* = FALSE, the sync word of the first XLL frame header is NOT present in the current frame and the decoder is not capable, when starting from non-synchronized state, of establishing synchronization with the data present in this frame. If the decoder is already in a synchronized state, it shall take *m_nuAudioChunksCumulSmoothPayload* bytes of payload from the current frame and place them into the smoothing buffer regardless of the value of *m_bACFirstXLLSyncPresent*.

6.4.16.5 nuVBRSmoothingBuffSzkB

nuVBRSmoothingBuffSzkB is present in the stream if *m_bVBRSmoothingBufferEnabled* = TRUE and *m_bACFirstXLLSyncPresent* = TRUE. *m_nuVBRSmoothingBuffSzkB* represents the size, in kBytes, of the variable bit rate smoothing buffer. The buffer size can take values of 64, 128, 192 and 256 kBytes and is constant for the entire duration of the stream. For DTS-UHD streams, the required buffer size is 128 kBytes (131 072 Bytes). It is assumed that this buffer exists in the decoder.

6.4.16.6 m_nuInitACXLLDecDlyFrames

m_nuInitACXLLDecDlyFrames is present in the stream if *m_bVBRSmoothingBufferEnabled* = TRUE and *m_bACFirstXLLSyncPresent* = TRUE. This parameter is the number of frames of the audio chunks payload that shall be buffered before the decoding can start. The payload is stored in the decoder's smoothing buffer and the offset is interpreted as a time stamp of the delay. The decoder receives the data, detects the offset number and does not decode a frame until *m_nuInitACXLLDecDlyFrames* frame intervals have elapsed. The decoder outputs shall stay muted until the buffered decoding can start. In subsequent frames, the decoder consumes the frame payload data from the smoothing buffer, outputs decoded audio and ignores the values of *m_nuInitACXLLDecDlyFrames* in every consecutive frame.

6.4.16.7 m_nuACFirstXLLSyncOffsInDWORDS

m_nuACFirstXLLSyncOffsInDWORDS is present in the stream if *m_bVBRSmoothingBufferEnabled* = TRUE and *m_bACFirstXLLSyncPresent* = TRUE. This value specifies the offset in 32-bit words from the start of the first audio chunk in the current frame payload to the start of an audio chunk that contains the first XLL frame header sync word. The start of the smoothed audio chunks frame payload is at the first 32-bit boundary immediately following the last metadata chunk as shown in Figure 6-1.

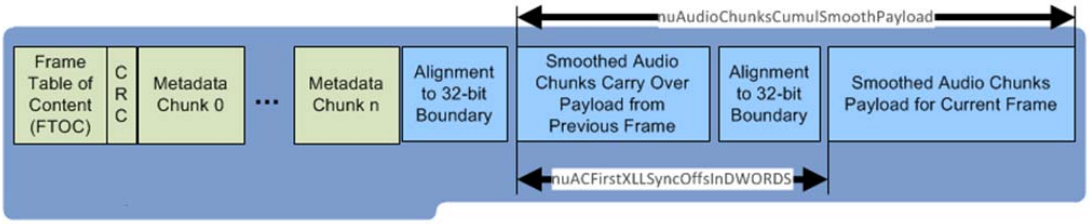

Figure 6-1: DTS-UHD Stream Structure

6.4.17 Reserved Fields

Skipping over the Byte Align bits and Reserved fields is shown in Table 6-27.

Table 6-27: SkipReservedandAlignFields

Syntax

```
// Function skips Reserved and Byte Alignment Fields
// Input Arguments:
// - bSyncFrameFlag if TRUE indicates that this is a sync frame
// - unBitCounter - state of the bit counter prior to extraction of FTOC sync word
// - unFTOCPayloadinBytes - size of FTOC in bytes starting from sync word to the FTOC CRC (if
present)
//
void CFrmTblOfContent::SkipReservedandAlignFields(bool bSyncFrameFlag, uint unBitCounter, uint
unFTOCPayloadinBytes)
{
  uint unBitsExtracted;
  // Calculate the number of extracted bits since the beginning of FTOC.
  // Bit counter at the beginning of FTOC is passed in as unBitCounter.
  unBitsExtracted = GetBitCounter() - unBitCounter;
  // Calculate number of bits for reserved and byte alignment fields
  if (bSyncFrameFlag) {
    // In sync frame subtract 2 bytes from FTOC payload for CRC
    unBitCounter = ((unFTOCPayloadinBytes-2)<<3) - unBitsExtracted;
  } else {
    unBitCounter = ((unFTOCPayloadinBytes)<<3) - unBitsExtracted;
  }
  // Skip unBitCounter bits occupied by reserved and byte alignment fields
  AdvanceBitReadPointer(unBitCounter);
}
```

6.4.18 FTOC CRC Word

This CRC value is calculated over the FTOC metadata fields, starting from the sync word (i.e. the sync word is included) up to and including the byte alignment field prior to the CRC word. The decoder shall search for *unSyncWord* on every DWORD aligned boundary in the expected search range, and after finding a possible one, it shall:

1) Extract *unFTOCPayloadinBytes*.

2) Calculate the CRC(16) value over the FTOC data fields (i.e. over *unFTOCPayloadinBytes*-2 bytes of data).

3) Extract the FTOC CRC field and compare it against the calculated CRC(16) value.

4) If the two values match, reverse back and start extracting the FTOC fields; otherwise, pronounce the false detection of a FTOC sync word, reverse back to the 32-bit boundary immediately after the falsely detected FTOC sync word and continue searching for a new *unSyncWord* pattern.

The above procedure is performed after the extraction of *m_bFullChannelBasedMixFlag* within the function for extraction of FTOC: Stream Parameters.

Specific implementation of CRC related functions is listed in clause 6.3.8.

7 Metadata Chunk (ID = 0x01)

7.1 Overview

The Metadata Chunk (Chunk ID = 0x01) is the primary structure for metadata descriptions for channel and object-based presentations. A Metadata Chunk can apply to an Object, an Object Group or a Channel-based Presentation. Multiple Metadata Chunks can apply to the same Audio Chunks in different Presentations.

7.2 Object Class Description and Handler Functions

The class description of C0x01MDChunkBody is shown in Table 7-1.

To minimize the overhead in metadata chunks, the full data is transmitted only in MD Sync Frames. In all other frames, denoted as the MD Predictive Frames, only the updates of the metadata are transmitted, and all the missing data is inherited from the values used in the previous frame.

This metadata chunk carries:

- the description for a multitude of objects and object groups;
- the overall presentation scaling parameters for different playback configurations;
- the static metadata that is distributed over multiple frames.

NOTE: When *bFullChannelBasedMixFlag* (extracted within FTOC) is TRUE, then all metadata within this chunk is static. Consequently when *bFullChannelBasedMixFlag* is TRUE, this 0x01 metadata chunk is transmitted ONLY in sync frames.

Table 7-1: Class Definition of C0x01MDChunkBody

Syntax

```
private:
  CBitStream *m_pBitStream;
  uchar m_ucAudPresIndex; // Audio Presentation Index
  uint m_unNumObjects; // Total number of objects and object groups

public:
  CObjListHandler   m_CObjList
  bool m_bMixStudioParamsPresent
  uchar m_ucRadiusRefernceUnitSphereInMeters
  uint m_unRefScreenHorizontalViewingAngleInDeg
  uchar m_ucRefScreenAspectRatio

// member functions
C0x01MDChunkBody (CBitStream *pBitStream, uint unmaxobjects)
void    Extract_Main_0x01_MDChunk (bool bSyncFrameFlag,
                                   bool bFullChannelBasedMixFlag,
                                   uint unBitCountBeforeMDID,
                                   uint unMDChunkSize,
                                   uint unFrameDuration,
                                   uint unOutChMask,
                                   bool bInteractObjLimitsPresent,
                                   CMFDStaticMD *pCMFDStaticMD,
                                   bool bAudioPresSelectableFlag,
                                   std::vector< uint > &punObjectIDList)
```

Syntax
void ExtractMetadataForObjects (bool bSyncFrameFlag, bool bFullChannelBasedMixFlag, uint unFrameDuration, uint unOutChMask, bool bInteractObjLimitsPresent, std::vector< uint > &punObjectIDList) bool CheckIfMDIsSuitableforImplObjRenderer (uint unObjectID) void ApplyExplObjList2ObjActMask (uint unExplObjListMask, std::vector< uint > &unObjectIDList) void ApplyGroupDefaultObjectActivityMask (void) uint TableLookUp (uint) // Function prototypes void SetNumOfObjects(uint unNumObjects){ m_unNumObjects = unNumObjects; } void SetAudPresIndex(uchar ucAudPresIndex){m_ucAudPresIndex = ucAudPresIndex;}

7.3 Functions Called from UnpackMDFrame

7.3.1 ApplyExplObjList2ObjActMask

ApplyExplObjList2ObjActMask (), shown in Table 7-2, overwrites object/object group activity flags based on the explicit object ID list. Up to 32 objects can be selected explicitly within a single metadata chunk.

Input Arguments:

- **unExplObjListMask** bit-mask indicates which object within this metadata chunk shall be activated where the LSB of the mask corresponds to the first packed object and so on in the packing order.
- **unObjectIDList** vector containing the list of the object ID whose metadata is present in current frame (in the packing order).

Note that static objects whose metadata is not transmitted in the current frame are not listed in either the *unExplObjListMask* or *unObjectIDList*. Their activity flags consequently keep their state from the last time the metadata was transmitted.

Table 7-2: ApplyExplObjList2ObjActMask

Syntax	Reference

void C0x01MDChunkBody::ApplyExplObjList2ObjActMask(uint unExplObjListMask,
std::vector<uint> & unObjectIDList)
{
uint unDesObjID;
uint unDesObjIndex;
bool bDesiredFlag;
 	
uint unPreObjCounter = 0;
for (uint nso = 0; nso < m_unNumObjects; nso++)
{
bDesiredFlag = ((unExplObjListMask >> nso) & 0x01) ? TRUE : FALSE;
// Get desired object ID corresponding to the nso-th packed object
unDesObjID = unObjectIDList[nso];
// Get the desired object index within the m_CObjList
unDesObjIndex = m_CObjList.FindObjIndex(unDesObjID);
// Set the activity flag of referenced object to the bDesiredFlag
if (unDesObjID < OBJGROUPIDSTART)
{
// Case of object ID corresponding to a 3D object
m_CObjList.m_pArrayOfObjPntrs[unDesObjIndex]->
m_pCLean3DMD->m_bObjActiveFlag = bDesiredFlag;
} else {
// Object ID corresponding to an object group
m_CObjList.m_pArrayOfObjPntrs[unDesObjIndex]->
m_pCGrpDef->m_bObjGrpActiveFlag = bDesiredFlag;
}
} // End of for (uint nso = 0; nso < m_unNumObjects; nso++)
} // End of ApplyExplObjList2ObjActMask()
	7.3.3

7.3.2 ApplyGroupDefaultObjectActivityMask

ApplyGroupDefaultObjectActivityMask(), shown in Table 7-3, overwrites the 3D object activity flags based on the default group object dependency masks. Notice that certain properties of the object group are used to unwind the object group hierarchy.

Both 3D objects and other object groups may be referenced within the object group metadata as long as they are all associated with (defined within) the same 0x01 metadata chunk, and the group hierarchy follows the packing order (i.e. only the objects/groups packed prior to the current object group can be referenced within this group). This allows for unwinding of the entire group hierarchy in a single loop as long as the loop is written in reverse order of packing as is done in this function.

Table 7-3: ApplyGroupDefaultObjectActivityMask

Syntax	Reference

`void C0x01MDChunkBody::ApplyGroupDefaultObjectActivityMask(void)`
`{`
`  uint unObjectId;`
`  uint unAffectedObjID;`
`  bool bDesiredFlag;`
`  int  nAffectedObjIndex;`
 	
`  for (uint nso = m_unNumObjects - 1; nso >= 0; nso--)`
`  {`
`    unObjectId = m_CObjList.m_punObjIDs[nso];`
`    // Check if object group`
`    if (unObjectId >= OBJGROUPIDSTART && unObjectId < 256)`
`    {`
`      // Check if group is active`
`      if (m_CObjList.m_pArrayOfObjPntrs[nso]->m_pCGrpDef->m_bObjGrpActiveFlag)`
`      {`
`        uchar ucNumObj = m_CObjList.m_pArrayOfObjPntrs[nso]->`
`                                        m_pCGrpDef->m_ucNumObjReferencedInGroup;`
`        uint unObjActMask = m_CObjList.m_pArrayOfObjPntrs[nso]->`
`                                            m_pCGrpDef->m_unWithinGrpObjActMask;`
`        uint *punObjIDList = m_CObjList.m_pArrayOfObjPntrs[nso]->`
`                                                m_pCGrpDef->m_punObjGrpRefIDs;`
`        // Loop over referenced objects within the group`
`        for (uint nor = 0; nor < ucNumObj; nor++)`
`        {`
`          // Check if nor indexed referenced object is active`
`          bDesiredFlag = ((unObjActMask >> nor) & 0x01) ? TRUE : FALSE;`
 	
`          // Get the ID of referenced object`
`          unAffectedObjID = punObjIDList[nor];`
 	
`          // Get the index of the referenced object`
`          nAffectedObjIndex = m_CObjList.FindObjIndex(unAffectedObjID);`
 	
`          // Set the activity flag of referenced object to the bDesiredFlag`
`          if (unAffectedObjID < OBJGROUPIDSTART)`
`            {`
`              // Case of object ID corresponding to a 3D object`
`              m_CObjList.m_pArrayOfObjPntrs[nAffectedObjIndex]->`
`                                      m_pCLean3DMD->m_bObjActiveFlag = bDesiredFlag;`
`          }`
`          else`
`          {`
`            // In this case the referenced object is another group`
`            // Notice that by definition all its referenced objects are packed prior`
`            // to this group i.e. their corresponding nso-s are smaller than the`
`            // current nso. So even if this group references other groups, hierarchy`
`            // is followed correctly.`
`            m_CObjList.m_pArrayOfObjPntrs[nAffectedObjIndex]->`
`                                     m_pCGrpDef->m_bObjGrpActiveFlag = bDesiredFlag;`
`          }`
`        } // End of for (uint nor = 0; nor < ucNumObj; nor++)`
`      } // End of if (m_CObjList.m_pArrayOfObjPntrs[nso]..`
`    } // End of if (unObjectId >= OBJGROUPIDSTART && unObjectId < 256)`
`  } // End of for (uint nso = m_unNumObjects - 1; nso >= 0; nso--)`
`} // End of ApplyGroupDefaultObjectActivityMask()`
	7.3.3

7.3.3 Find Object Index

This function returns an index into the list of objects with the ID given by *unDesiredObjID*.

Syntax

```
int CObjListHandler::FindObjIndex(uint unDesiredObjID) {
  // Check list up to the highest occupied index
  for (uint unidex = 0; unidex<m_unHigestIndex; unidex++)
  {
    if (unDesiredObjID == m_punObjIDs[unidex]) {
      // Object is in the list at location defined by unidex
      return unidex;
    }
  }
  // Since object ID was not found in the list return -1
  return -1;
}
```

7.4 Extract_Main_0x01_MDChunk

This function extracts the parameters for the Main Body of the 0x01 Metadata Chunk, as shown in the pseudo-code example shown in Table 7-4.

Input parameters:

- *bSyncFrameFlag* is a flag set during extraction of a Frame Table Of Content (FTOC). The bSyncFrameFlag is TRUE if and only if the FTOC SYNC WORD is present in this frame.
- *bFullChannelBasedMixFlag* is the flag extracted within the FTOC. When the bFullChannelBasedMixFlag is TRUE and the bSyncFrameFlag=FALSE this metadata chunk is not transmitted.
- *unBitCountBeforeMDID* - bit counter prior to extraction of MD Chunk ID.
- *unMDChunkSize* - size of the metadata chunk in Bytes including the MD Chunk ID and CRC (if present).
- *unFrameDuration* - frame duration in clock cycles.
- *unOutChMask* - channel mask defining the desired output channel layout. This bit mask has the same channnel mask assignment as the 32-bit *ChAvtivityMask* defined in clause 7.8.11.14.2.
- *bInteractObjLimitsPresent* - flag indicating if custom limits on the user interactions are carried in the stream.
- **pCMFDStaticMD* Pointer to a class with static MFD metadata.
- *bAudioPresSelectableFlag* - flag indicating if this chunks belongs to a selectable audio presentation.
- *punObjectIDList* - list of object IDs described in this metadata chunk in the order of packing.

Table 7-4: ExtractandInvQuantParams

Syntax	Reference

`void C0x01MDChunkBody:: Extract_Main_0x01_MDChunk (bool bSyncFrameFlag,`
`bool bFullChannelBasedMixFlag,`
`uint unBitCountBeforeMDID,`
`uint unMDChunkSize,`
`uint unFrameDuration,`
`uint unOutChMask,`
`bool bInteractObjLimitsPresent,`
`CMFDStaticMD *pCMFDStaticMD,`
`CPresScaling *pCPresScaling,`
`bool bAudioPresSelectableFlag,`
`std::vector<uint> & punObjectIDList)`
`{`
`if (bAudioPresSelectableFlag) {`
`if (bSyncFrameFlag){`
`if (pCPresScaling == NULL)`
`{`
`// CPresScaling class holds presentation scaling parameters`
	7.5.1

Syntax	Reference

pCPresScaling = new CPresScaling(m_pBitStream);
}
pCPresScaling->ExtractPresScalingParams(bFullChannelBasedMixFlag);
// Is Static Metadata Distributed Over Multiple Frames is present
m_bMFDistrStaticMDPresent = (ExtractBits(1) == 1) ? TRUE : FALSE;
}
	7.5.3

if (m_bMFDistrStaticMDPresent) {
if (pCMFDStaticMD == NULL)
{
// CMFDStaticMD class holds static metadata parameters
pCMFDStaticMD = new CMFDStaticMD(m_pBitStream);
}
// Extract static metadata payload
pCMFDStaticMD->ExtractMultiFrameDistribStaticMD(bSyncFrameFlag);
}
	7.6

}
else {
m_bMFDistrStaticMDPresent = FALSE;
}	

// Extract Object and Object Group Definitions
ExtractMetadataForObjects (
bSyncFrameFlag,
bFullChannelBasedMixFlag,
unFrameDuration,
unOutChMask,
bInteractObjLimitsPresent,
punObjectIDList);
// Calculate the total number of extracted bits in this chunk
uint unExtractedBits = GetBitCounter() - unBitCountBeforeMDID;
 	
// Skip over the Reserved, Byte Alignment and CRC Fields
AdvanceBitReadPointer(unMDChunkSize*8 - unExtractedBits);
 	
} // End of Extract_Main_0x01_MDChunk
	7.8

7.5 Functions Called from Extract_Main_0x01_MDChunk

7.5.1 Presentation Scaling Parameters

This metadata block is transmitted only in the sync frames for audio presentations with *bAudPresSelectableFlag* set to TRUE. These parameters allow for additional scaling of the overall output level and shall be applied during object rendering. Up to four distinct scaling parameters can be transmitted, corresponding to the 2.x, 5.x, 7.x and 11.x output layouts.

The decoder shall use these parameters only if *m_bOutScalesDerivedfromBitStream* is TRUE. If the desired output layout corresponds to one of the provided layouts, the decoder shall use that set of parameters. If the desired output layout does not match one of the prescribed patterns, the pattern with the next largest number of full scale channels shall be used. For example if the output layout is 3.1, 4.1 or 5.1, the decoder will apply the scale that corresponds to a scale factor designated for 5 full bandwidth channels (i.e. *m_unOutScale5px*).

When the multiple audio presentations are being decoded, the same scaling parameter selection criteria shall be used. If there are multiple scaling parameters that satisfy this condition, then the one corresponding to the highest presentation index (*m_ucAudPresIndex*) shall be used.

If *m_bOutScalesDerivedfromBitStream* is FALSE, then no presentation scaling parameters were transmitted for this presentation. The decoder shall not use the default values of the presentation scaling parameters (*m_unOutScale2px*, *m_unOutScale5px*, *m_unOutScale7px* or *m_unOutScale11px*) but rather shall calculate the presentation scaling factor.

Pseudo-code implementing the extraction of the presentation scaling parameters is listed in Table 7-5.

Table 7-5: ExtractPresScalingParams

Syntax	Reference

`void CPresScaling::ExtractPresScalingParams(bool bFullChannelBasedMixFlag)`
`{`
`  uchar ucTable[4] = { 2, 5, 7, 11 }; // Table with number of full bandwidth channels`
`                                    // associated with m_unOutScale.`
`  bool bOutScalePresent;`
`  int nTemp;`
`  uint unScales[4];`
`  float rTemp;`
 	
`  m_bOutScalesDerivedfromBitStream = false;`
`  for (uint nn = 3; nn ≥ 0; nn--)`
`  {`
`    // Flag indicating presence of a scale factor for output layouts with`
`    // NumCh = 2, 5, 7 or 11 channels corresponding to nn=0,1,2, or 3 respectively`
`        //`
`    bOutScalePresent = (ExtractBits(1) == 1) ? TRUE : FALSE;`
	7.5.2.1

`    // Scale factor for output layouts with NumCh full bandwidth channels: in range`
`    // from -25dB to +6dB`
`    // Final values are translated to linear scale in Q3.15 fixed point format.`
`    if (bOutScalePresent){`
`      // Extract 5-bit gain table look-up index and translate it to`
`      // range -19dB to +12dB`
`      nTemp = -25 + ((int)ExtractBits(5));`
`      // Translate to linear scale`
`      rTemp = pow(DTS_FLP_TEN, ((float)nTemp) / DTS_FLP_TWENTY);`
`      // Translate to Q2.15 fixed point format;`
`      // unScales[nn] = (uint)floor(rTemp * ((float)(1 << 15)) + (float)(1 << 14));`
`    }`
`    else{`
`      unScales[nn] = (nn == 3) ? ONE_IN_Q15 : unScales[nn+1];`
`    }`
`    m_bOutScalesDerivedfromBitStream = (m_bOutScalesDerivedfromBitStream) ? true :`
`                                                                    bOutScalePresent`
`  } // End of for (uint nn=3; …`
 	
`  // Save the extracted scale values`
`  m_unOutScale2px = unScales[0];`
`  m_unOutScale5px = unScales[1];`
`  m_unOutScale7px = unScales[2];`
`  m_unOutScale11px = unScales[3];`
`} // End of CPresScaling::ExtractPresScalingParams()`
	7.5.2.2

7.5.2 Parameters for ExtractPresScalingParams

7.5.2.1 bOutScalePresent

This 1-bit flag is transmitted in the stream only if *bFullChannelBasedMixFlag* is FALSE. *bOutScalePresent* is a flag that indicates whether an audio presentation scaling parameter is transmitted for particular output configuration.

When the *bFullChannelBasedMixFlag* is TRUE, *bOutScalePresent* is by default set to FALSE.

7.5.2.2 Presentation Scale

This 5-bit field is transmitted in the stream only if *bOutScalePresent* is TRUE. The value corresponds to the scale factor in range from -19 dB to +12 dB with a step size of 1 dB.

7.5.3 m_bMFDistrStaticMDPresent

This field is transmitted only for selectable audio presentations (*bAudioPresSelectableFlag* is TRUE) and only in the sync frames (*bSyncFrameFlag* is TRUE). If *m_bMFDistrStaticMDPresent* is TRUE then the static metadata distributed over multiple frames is present in the stream.

For non-selectable audio presentations the *m_bMFDistrStaticMDPresent* is by default set to FALSE.

In frames where the *bSyncFrameFlag* is FALSE the *m_bMFDistrStaticMDPresent* maintains its value from the last sync frame.

7.6 Static Metadata Distributed Over Multiple Frames

A block of global static metadata is subdivided into packets of data transmitted over *m_unNumStaticMDPackets* frames. The Static Metadata Frame class is shown in Table 7-6. Each packet is a uniform size defined by *m_unStaticMDPacketByteSize*. Note that the last packet may be zero padded to force it to the correct number of bytes.

A sequence of packets always starts at a sync frame and will be completed before the next sync frame. Parameters that control the distribution of payload over multiple packets are transmitted only in the sync frames. Packets are transmitted in each frame over the period of *m_unNumStaticMDPackets* consecutive frames.

Pseudo-code for extraction of parameters that control the distribution of metadata packets and assembling of packets into single metadata block can be found in Table 7-7.

Table 7-6: Static Metadata Frame Class

Syntax

```
class CMFDStaticMD
{
// Class for static metadata whose transmission is distributed over multiple frames
private:
  CBitStream *m_pBitStream;
  CBitStream *m_pStaticMDBitStream;
  uint m_unNumStaticMDPackets;
  uint m_unStaticMDPacketByteSize;    // Size of each packet of static metadata
  uint m_unNumPacketsAcquired;        // Tracks the number of acquired packets
  uint m_bStaticMetadataUpdtFlag;     // Indicates if the static metadata has changed
  uint m_punStaticLDMetadata_AllocSize; // Allocated size of m_pStaticMDBitStream
  bool m_bStaticMDParamsExtracted;
  bool m_bNominalLD_DescriptionFlag;  // nominal or extended L&D metadata description
  uint m_unNumDRCGrps;                 // Number of DRC groups described in static MD
  CMatrix<CDRCProfile> m_CDRCProfiles;
  uchar m_ucNumLongTermLoudnessMsrmSets; //
  std::vector<CLTLoudnMsrmSet> m_CLTLoudnMsrmSet_Vector;
  bool m_bIsLTLoudnMsrsmOffLine;

public:

…
}; // End of CMFDStaticMD class definition
```

Table 7-7: ExtractMultiFrameDistribStaticMD

Syntax	Reference

`void CMFDStaticMD::ExtractMultiFrameDistribStaticMD(bool bSyncFrameFlag,`
`  bool bFullChannelBasedMixFlag)`
`{`
`  if (bSyncFrameFlag) {`
`  m_unNumPacketsAcquired = 0;`
`  if (bFullChannelBasedMixFlag) {`
`  m_unNumStaticMDPackets = 1;`
`  }`
`  else {`
`  uchar uctable[4] = { 0, 6, 9, 12 };`
`  m_unNumStaticMDPackets = (uint)ExtractVarLenBitFields(uctable) + 1;`
`  }`
`  if (bFullChannelBasedMixFlag) {`
`  m_pStaticMDBitStream = m_pBitStream;`
`  }`
`  else {`
`  uchar uctable[4] = { 5, 7, 9, 11 };`
`  m_unStaticMDPacketByteSize = (uint)ExtractVarLenBitFields(uctable) + 3;`
`  if (m_pStaticMDBitStream == NULL) {`
`  m_punStaticLDMetadata_AllocSize =`
`  m_unNumStaticMDPackets*m_unStaticMDPacketByteSize;`
`  m_pStaticMDBitStream = new CBitStream(m_punStaticLDMetadata_AllocSize);`
`  }`
`  else {`
`  uint unReqSizeInDWords = m_unNumStaticMDPackets*m_unStaticMDPacketByteSize;`
	
	7.7.1
	
	7.7.2

Syntax	Reference

`        if (m_punStaticLDMetadata_AllocSize < unReqSizeInDWords) {`
`          // Resize the buffer within the m_pStaticMDBitStream`
`          delete m_pStaticMDBitStream;`
`          m_punStaticLDMetadata_AllocSize = unReqSizeInDWords;`
`          m_pStaticMDBitStream = new CBitStream(m_punStaticLDMetadata_AllocSize);`
`        }`
`      }`
`      // Make sure buffer is filled with all zeros`
`      memset(m_pStaticMDBitStream->GetPntrToTopOfDataBuffer(), 0,`
`m_punStaticLDMetadata_AllocSize*sizeof(uint));`
`    } // End of case when the bFullChannelBasedMixFlag is FALSE`
`    if (m_unNumStaticMDPackets > 1) {`
`      m_bStaticMetadataUpdtFlag = (ExtractBits(1) == 1) ? TRUE : FALSE;`
	7.7.3

`    }`
`    else {`
`      m_bStaticMetadataUpdtFlag = TRUE;`
`    }`
`  } // End of case when the bSyncFrameFlag is TRUE`
`  if (m_unNumPacketsAcquired < m_unNumStaticMDPackets) {`
`    if (bFullChannelBasedMixFlag == FALSE) {`
`      uint unByteOffs = m_unNumPacketsAcquired*m_unStaticMDPacketByteSize;`
`      uchar *pucTmp = m_pStaticMDBitStream->GetPntrToTopOfDataBuffer() + unByteOffs;`
`      for (uint nw = 0; nw < m_unStaticMDPacketByteSize; nw++) {`
	7.7.4

`        *pucTmp++ = (uchar)ExtractBits(8);`
`      }`
`    }`
`    // Update acquired packet counter`
`    m_unNumPacketsAcquired = m_unNumPacketsAcquired + 1;`
`    if (m_unNumPacketsAcquired == m_unNumStaticMDPackets) {`
`      // All static metadata has been acquired`
`      if (m_bStaticMetadataUpdtFlag || (m_bStaticMDParamsExtracted == FALSE)) {`
`        // If there is an update of static metadata or if the parameters`
`        // have never been extracted, extract the L&D Parameters.`
`        const bool bExtractOnlyFirstLTLMeasurementSet = FALSE;`
`        ExtractStaticLandDParams(bFullChannelBasedMixFlag,`
	7.7.5

`                                                 bExtractOnlyFirstLTLMeasurementSet);`
`      }`
`    }`
`    else if (m_unNumPacketsAcquired == 1) {`
`      if (m_bStaticMetadataUpdtFlag || (m_bStaticMDParamsExtracted == FALSE))`
`      {`
`        const bool bExtractOnlyFirstLTLMeasurementSet = TRUE;`
`        ExtractStaticLandDParams(bFullChannelBasedMixFlag,`
	7.7.5

`                                                 bExtractOnlyFirstLTLMeasurementSet);`
`      }`
`    }`
`  } // End of if (m_unNumPacketsAcquired < m_unNumStaticMDPackets)`
`} // End of CMFDStaticMD::ExtractMultiFrameDistribStaticMD()`	

7.7 Parameters for ExtractMultiFrameDistribStaticMD

7.7.1 m_unNumStaticMDPackets

This variable length field is transmitted in the stream only if *bFullChannelBasedMixFlag* is FALSE. Its value indicates the number of packets/frames needed to assemble entire static metadata block. This parameter is transmitted as a variable length code with length of 1, 8, 12 or 15 bits allowing for 1, 65, 577 or 4 673 packets respectively.

When *bFullChannelBasedMixFlag* is TRUE, all static metadata parameters are transmitted in the sync frame hence *m_unNumStaticMDPackets* is 1 by default.

7.7.2 m_unStaticMDPacketByteSize

This variable length field is transmitted in the stream only if the *bFullChannelBasedMixFlag* is FALSE. Its value indicates the size in Bytes of each packet of static metadata. The *m_unStaticMDPacketByteSize* is transmitted as a variable length code with length of 6, 9, 12 or 14 bits allowing for 34, 162, 674 or 2 722 Bytes respectively.

Note that the minimum value for *m_unStaticMDPacketByteSize* is 3 Bytes. This guarantees that an integrated loudness parameter set for the complete audio presentation (always packed first within the static metadata payload) is fully contained within the first distributed metadata packet. Consequently, the decoder can start performing overall loudness correction immediately after establishing the synchronization and without waiting for the rest of distributed metadata packets to arrive.

In case when the *bFullChannelBasedMixFlag* is TRUE, the size of the static metadata block is not transmitted explicitly. Extraction of the static metadata parameters shall be performed one by one, directly from the 0x01 metadata stream buffer. Note that, in this case, no reserved or byte alignment fields are present at the end of the static metadata block.

7.7.3 m_bStaticMetadataUpdtFlag

This flag indicates if there is an update of static metadata content since the previous distributed transmission. *m_bStaticMetadataUpdtFlag* is transmitted only in the sync frame.

After assembling the entire static metadata buffer, decoder shall check:

- If the *m_bStaticMetadataUpdtFlag* is TRUE, then the decoder unpacks the static metadata and overwrites the existing values in the corresponding parameter buffers.
- If the *m_bStaticMetadataUpdtFlag* is FALSE and the static metadata HAS NOT BEEN unpacked previously, decoder unpacks the static metadata into corresponding parameter buffers.
- If the *m_bStaticMetadataUpdtFlag* is FALSE and the static metadata HAS BEEN unpacked previously, the the decoder skips the unpacking of the static metadata buffer and the L&D processing/control module is responsible for keeping the existing values in its parameter buffers.

7.7.4 Metadata Packet Payload

A packet payload extracted in current frame shall be placed into the appropriate location within the static metadata *buffer m_pStaticMDBitStream*.

For the syntax of metadata captured in the *m_pStaticMDBitStream* refer to clause 7.7.5.

7.7.5 Static Loudness and Dynamics Metadata

Static L&D metadata (due to its static nature) can be carried as a distributed payload over number of encoded frames. Static metadata contains multiple Loudness and Dynamics profiles, each describing the configuration parameters of the dynamic range compressor that is to be used with a particular decoder-selected compression type. In addition, various types of long term loudness measures are also carried as a part of this metadata block.

Pseudo-code for extraction of static loudness and dynamics parameters from the static metadata block can be found in Table 7-8.

Table 7-8: ExtractStaticLandDParams

Syntax	Reference

`uint CMFDStaticMD::ExtractStaticLandDParams(bool bFullChannelBasedMixFlag,`
`  bool bExtractOnlyFirstLTLMeasurementSet)`
`{`
`  uint unBitCounter;`
`  if (bFullChannelBasedMixFlag == FALSE) {`
`  m_bNominalLD_DescriptionFlag = (bool) m_pStaticMDBitStream->ExtractBits(1);`
`  }`
`  else {`
`  m_bNominalLD_DescriptionFlag = TRUE;`
`  }`
`  // Long term integrated loudness measurement parameters`
`  // Number of long-term loudness measure parameter sets`
`  if (m_bNominalLD_DescriptionFlag) {`
`  if (bFullChannelBasedMixFlag == FALSE) {`
`  m_ucNumLongTermLoudnessMsrmSets = (m_pStaticMDBitStream->ExtractBits(1)==0)?1 : 3;`
`  else {`
`  m_ucNumLongTermLoudnessMsrmSets = 1;`
	7.7.6.1
	7.7.6.2

Syntax	Reference

`    }`
`  }`
`  else {`
`    m_ucNumLongTermLoudnessMsrmSets = (uchar) m_pStaticMDBitStream->ExtractBits(4) + 1;`
	7.7.6.3

`  }`
`  // Long Term Loudness Measure Parameter Sets (different types)`
`  if (m_CLTLoudnMsrmSet_Vector.size() < m_ucNumLongTermLoudnessMsrmSets) {`
`    m_CLTLoudnMsrmSet_Vector.clear();`
`    m_CLTLoudnMsrmSet_Vector.resize(m_ucNumLongTermLoudnessMsrmSets,`
`CLTLoudnMsrmSet(m_pStaticMDBitStream));`
`  }`
`  uint unNumSetstoExtract = (bExtractOnlyFirstLTLMeasurementSet) ?`
`                                                1 : m_ucNumLongTermLoudnessMsrmSets;`
`  for (uchar npar = 0; npar < unNumSetstoExtract; npar++) {`
`    // Extract one LT loudness measurement set`
`    m_CLTLoudnMsrmSet_Vector[npar].ExtractLTLMParamSet(TRUE,`
`                                                m_bNominalLD_DescriptionFlag, npar);`
`  } // End of for (npar=0; npar<ucNumLongTermLoudnessParmSets; npar++)`
	7.7.6.4

`  if (bExtractOnlyFirstLTLMeasurementSet) {`
`    // Return after extracting the first LTL measurement set`
`    return 0;`
`  }`
`  if (m_bNominalLD_DescriptionFlag == FALSE) {`
`    m_bIsLTLoudnMsrsmOffLine = (bool) m_pStaticMDBitStream->ExtractBits(1);`
`  }`
	7.7.6.6

`  // DRC Parameters`
`  // Set the number of DRC groups described in the stream`
`  m_unNumDRCGrps = 1; //  Only 1 group defined corresponding to the full presentation`
`  const uint ng = 0;  // Fixed group index 0`
`  if (m_CDRCProfiles.m_ppMtrx == NULL) {`
`    // Allocate memory for a matrix of DRC profile classes`
`    m_CDRCProfiles.NewMatrix(NUMDRCOMPRTYPES, m_unNumDRCGrps);`
`  }`
`  // DRC Compression parameters for each of the DRC_COMPRESSION_TYPES.`
`  for (uchar nc = 0; nc < NUMDRCOMPRTYPES; nc++) {`
`    m_CDRCProfiles.m_ppMtrx[nc][ng].m_ucAssetType =`
`                                                ASSET_TYPE_COMPLETE_AUDIO_PRESENTATION`
`    m_CDRCProfiles.m_ppMtrx[nc][ng].m_bCustomDRCCurveMDPresent =`
`                                        (bool) m_pStaticMDBitStream->ExtractBits(1);`
`    if (m_CDRCProfiles.m_ppMtrx[nc][ng].m_bCustomDRCCurveMDPresent == TRUE) {`
`      ExtractCustomDRCCurves(&m_CDRCProfiles.m_ppMtrx[nc][ng]);`
`    }`
`    m_CDRCProfiles.m_ppMtrx[nc][ng].m_bCustomDRCSmoothMDPresent =`
`                          (m_pStaticMDBitStream->ExtractBits(1) == 1) ? TRUE : FALSE;`
`    if (m_CDRCProfiles.m_ppMtrx[nc][ng].m_bCustomDRCSmoothMDPresent == true) {`
`      // Get fast attack time constant`
`      m_CDRCProfiles.m_ppMtrx[nc][ng].m_rFastAttack =`
`                        rDRCFastAttackTable[m_pStaticMDBitStream->ExtractBits(6)];`
`      // Get slow attack time constant by 10 * the value from the fast attack table`
`      m_CDRCProfiles.m_ppMtrx[nc][ng].m_rSlowAttack =`
`                10.0f * rDRCFastAttackTable[m_pStaticMDBitStream->ExtractBits(6)];`
`      // Get fast release time constant`
`      m_CDRCProfiles.m_ppMtrx[nc][ng].m_rFastRelease =`
`                       rDRCFastReleaseTable[m_pStaticMDBitStream->ExtractBits(6)];`
`      // Get slow release time constant by 2 * the value from the fast release table`
`      m_CDRCProfiles.m_ppMtrx[nc][ng].m_rSlowRelease =`
`                2.0f * rDRCFastReleaseTable[m_pStaticMDBitStream->ExtractBits(6)];`
`      // Get slow to fast attack threshold`
`      m_CDRCProfiles.m_ppMtrx[nc][ng].m_rAttackThreshld = (float) 1.0 /`
`          rDRCSlow2FastThrshld_Table[(uint)m_pStaticMDBitStream->ExtractBits(6)]);`
`      // Get slow to fast release threshold`
`      m_CDRCProfiles.m_ppMtrx[nc][ng].m_rReleaseThreshld =`
`           rDRCSlow2FastThrshld_Table[(uint)m_pStaticMDBitStream->ExtractBits(6)];`
`    }  // End of if  (m_CDRCProfiles.m_ppMtrx … m_bCustomDRCSmoothMDPresent == TRUE`
`  }  // End of Compression Type Loop`
	7.7.6.7
	7.7.6.8
	7.7.6.10
	7.7.6.11
	7.7.6.12
	7.7.6.13
	7.7.6.14
	7.7.6.15
	7.7.6.16

`  if (bFullChannelBasedMixFlag == FALSE)`
`  {`
`    unBitCounter = m_pStaticMDBitStream->GetBitCounter() - unBitCounter;`
`    m_pStaticMDBitStream-`
`>AdvanceBitReadPointer(m_unNumStaticMDPackets*m_unStaticMDPacketByteSize * 8 -`
`unBitCounter);`
`  }`
`  m_bStaticMDParamsExtracted = TRUE;`
`  return 0;`
`} // End of CMFDStaticMD::ExtractStaticLandDParams()`
	7.7.6.17

7.7.6 Parameters for ExtractStaticLandDParams

7.7.6.1 m_bNominalLD_DescriptionFlag

This 1-bit field is transmitted in the stream only if *bFullChannelBasedMixFlag* is FALSE. This flag, if TRUE, indicates that a nominal (restricted feature) description of the loudness and dynamics metadata is used. If the *m_bNominalLD_DescriptionFlag* is FALSE, it indicates that an extended description of the loudness and dynamics metadata is used.

In case when the *bFullChannelBasedMixFlag* is TRUE the nominal L&D metadata description is used by default.

7.7.6.2 Integrated Loudness Parameters

Multiple sets of integrated loudness parameters may be transmitted for each audio presentation. When *m_bNominalLD_DescriptionFlag* is TRUE for a given set, it correspond to the loudness measurement of one of the following:

- A complete audio presentation.
- A complete audio presentation excluding all speech objects.
- All speech (dialog) objects rendered together.

In extended use cases (*m_bNominalLD_DescriptionFlag* is FALSE) an additional subdivision of complete audio presentation is allowed according to the predefined asset types.

7.7.6.3 m_ucNumLongTermLoudnessMsrmSets

This field is transmitted in the stream only if the *bFullChannelBasedMixFlag* is FALSE. It represents the number of long-term loudness measure parameter sets that are transmitted within the static metadata block. Valid range is:

- 1 or 3 if *m_bNominalLD_DescriptionFlag* is TRUE; or
- from 1 to 16 if *m_bNominalLD_DescriptionFlag* is FALSE.

If *m_bNominalLD_DescriptionFlag* is TRUE and *m_ucNumLongTermLoudnessMsrmSets*=1, then the loudness measurement for the complete audio presentation is transmitted.

If *m_bNominalLD_DescriptionFlag* is TRUE and *m_ucNumLongTermLoudnessMsrmSets* = 3, then the loudness measurement for:

- the complete audio presentation;
- the complete audio presentation excluding all speech objects; and
- all speech objects rendered together

are transmitted in the stream. In case when the *bFullChannelBasedMixFlag* is TRUE the number of long term loudness measurements sets is by default set to 1.

7.7.6.4 Long Term Loudness Measure Parameter Set

This function takes 3 arguments:

- *bLongTermLMPresent* - flag indicating if the Long Term Loudness Measure Parameter Set (LTLMPS) is transmitted in the stream.
- *m_bNominalLD_DescriptionFlag* - flag indicating if nominal (TRUE) or extended (FALSE) L&D metadata description is used.
- *unPackIndex* - packing index within the static metadata block corresponding to this integrated loudness measurement set.

The pseudo-code for extraction of this metadata block can be found in Table 7-9.

Table 7-9: ExtractLTLMParamSet

Syntax	Reference

`void CLTLoudnMsrmSet::ExtractLTLMParamSet(bool bLongTermLMPresent,`
`  bool bNominalLD_DescriptionFlag,`
`  uint unPackIndex)`
`{`
`  if (bLongTermLMPresent) {`
`  m_rLoudness = rLongTermLoudnessMeasure_Table[(uint) ExtractBits(6)];`
	7.7.6.5.1

`  if (bNominalLD_DescriptionFlag)`
`  {`
`  // If bNominalLD_DescriptionFlag is TRUE, the associated asset type is deduced`
`  // from the packing index`
`  switch (unPackIndex)`
`  {`
`  case 0:`
`  m_ucAssociatedAssetType = ASSET_TYPE_COMPLETE_AUDIO_PRESENTATION;`
`  break;`
`  case 1:`
`  m_ucAssociatedAssetType = ASSET_TYPE_COMPLETE_DIALOG;`
`  break;`
`  case 2:`
`  m_ucAssociatedAssetType =`
`  ASSET_TYPE_COMPLETE_AUDIO_PRES_EXCLUDING_DIALOG;`
`  break;`
`  default: // This would be an error condition`
`  }`
`  }`
`  else`
`  {`	

`  m_ucAssociatedAssetType = (uint)ExtractBits(5);`	7.7.6.5.2

`  }`
`  // Long Term Loudness Measurement Type used to calculate m_rLoudness.`
`  uchar ucBitWidth = (bNominalLD_DescriptionFlag) ? 2 : 4;`
`  m_ucLoudnessMsrmType = (uchar)ExtractBits(ucBitWidth);`
	7.7.6.5.3

`  } // End of bLongTermLMPresent is TRUE case`
`  else`
`  {`
`  m_rLoudness = (float) -24.0; // Default is -24.0dB`
`  m_ucAssociatedAssetType =`
`  (uchar) ASSET_TYPE_COMPLETE_AUDIO_PRESENTATION;`
`  m_ucLoudnessMsrmType = (uchar) LT_LOUDNESS_MEASUREMENT_TYPE_UNKNOWN;`
`  }`
`} // End of CLTLoudnMsrmSet::ExtractLTLMParamSet()`	

7.7.6.5 Parameters for ExtractLTLMParamSet

7.7.6.5.1 m_rLoudness

This 6-bit field is present in the stream only if the *bLongTermLMPresent* is TRUE. The extracted field is in the range of 0 to 63 and represents an index into Long Term Loudness Measure Table (clause 5.2.2.2). If *bLongTermLMPresent* is FALSE the *m_rLoudness* is set by default to -24,0 dB.

7.7.6.5.2 m_ucAssociatedAssetType

This variable length field is present in the stream only if it is part of a static metadata block.

The *m_ucAssociatedAssetType* is an index into Table 7-10. The 4-bit or 6-bit codes are used to extract the *m_ucAssociatedAssetType* in range from 0 to 7 or from 0 to 31 respectively.

If not transmitted the *m_ucAssociatedAssetType* is by default set to complete audio presentation type.

The long-term loudness measure (*m_rLoudness*) is associated with the asset type indicated by the *m_ucAssociatedAssetType*. The *m_rLoudness* corresponds to the joint long-term loudness measure of all active objects, of this asset type, within this audio presentation.

Table 7-10: Definition of Object Types

index	m_ucAssociatedAssetType
0	ASSET_TYPE_UNKNOWN
1	ASSET_TYPE_COMPLETE_AUDIO_PRESENTATION
2	ASSET_TYPE_COMPLETE_DIALOG
3	ASSET_TYPE_COMPLETE_AUDIO_PRES_EXCLUDING_DIALOG
4	ASSET_TYPE_BED_MIX_WITH_DIALOG
5	ASSET_TYPE_BED_MIX_EXCLUDING_DIALOG
6	ASSET_TYPE_DIALOG
7	ASSET_TYPE_MUSIC
8	ASSET_TYPE_EFFECTS
9	ASSET_TYPE_MUSIC_EFFECTS
10	ASSET_TYPE_COMMENTARY
11	ASSET_TYPE_VISUALLY_IMPAIRED
12	ASSET_TYPE_HEARING_IMPAIRED
13	ASSET_TYPE_AMBIENCE
14	ASSET_TYPE_ISOLATED_FOLEY
15	ASSET_TYPE_KARAOKE
16	ASSET_TYPE_NON_DIEGETIC
17	ASSET_TYPE_COMPOSITE_MULTI_SRC
18	ASSET_TYPE_NEARFIELD_BED
19	ASSET_TYPE_SPOKEN_SUBTITLE
20 - 31	Reserved

7.7.6.5.3 m_ucLoudnessMsrmType

This field is present in the stream only if *bLongTermLMPresent* is TRUE. If *bNominalLD_DescriptionFlag* is TRUE, then this field is 2 bits, otherwise it is 4 bits. In both cases, the value of *m_ucLoudnessMsrmType* is used as an index into Table 7-11 and indicates a type of long term loudness measurement that was used for calculation of the *m_rLoudness*.

If *bLongTermLMPresent* is FALSE, then *m_ucLoudnessMsrmType* is set to the 'Unknown' type.

Table 7-11: Long Term Loudness Measurement Types

Index	Loudness Measurement Types	Mnemonic
0	Unknown	LT_LOUDNESS_MEASUREMENT_TYPE_UNKNOWN
1	Manual	LT_LOUDNESS_MEASUREMENT_TYPE_MANUAL
2	ATSC A85 [i.1]	LT_LOUDNESS_MEASUREMENT_TYPE_A85
3	EBU R128 [i.2]	LT_LOUDNESS_MEASUREMENT_TYPE_EBU_R128
4	BS.1770 (see note 1)	LT_LOUDNESS_MEASUREMENT_TYPE_1770
5	BS.1770 Dialogue Gate (see note 2)	LT_LOUDNESS_MEASUREMENT_TYPE_1770_DG
6	BS.1770 Non-Dialogue Gate (see note 3)	LT_LOUDNESS_MEASUREMENT_TYPE_1770_NDG
7 - 15	Reserved	LT_LOUDNESS_MEASUREMENT_TYPE_RESERVED
NOTE 1: According to Equation 7 of Recommendation ITU-R BS.1770-4 [2]. NOTE 2: According to Equation 2 of Recommendation ITU-R BS.1770-4 [2] that is only active during dialog components of the signal. NOTE 3: According to Equation 2 of Recommendation ITU-R BS.1770-4 [2] that is only active during non-dialog components of the signal.		

7.7.6.6 m_bIsLTLoudnMsrsmOffLine

This 1-bit flag is transmitted in the stream only if the *m_bNominalLD_DescriptionFlag* is FALSE. Its value indicates if the process used to encode the long term loudness measurements was running:

- in real time (*m_bIsLTLoudnMsrsmOffLine* = 0); or
- as an off-line/multi-pass process (*m_bIsLTLoudnMsrsmOffLine* = 1).

7.7.6.7 m_bCustomDRCCurveMDPresent

A separate *m_bCustomDRCCurveMDPresent* flag is sent for every DRC compression type shown in the Definition of DRC Compression Types in Table 7-12.

If *m_bCustomDRCCurveMDPresent* is TRUE, then for a particular DRC compression type, a custom DRC compression curve is transmitted in the stream. Otherwise, the decoder's default compression curve is used for the particular DRC compression type.

Table 7-12: DRC Compression Types

DRC Compression Type	Associated DRC Group Type
0	DRC_COMPRESSION_TYPE_LOW_FULL
1	DRC_COMPRESSION_TYPE_MEDIUM_FULL
2	DRC_COMPRESSION_TYPE_HIGH_FULL

7.7.6.8 Custom DRC Curves

A custom DRC curve may be transmitted for each of the DRC compression types. A curve may be described exclusively by an index (*m_ucDRCCurveIndex*) into a table of predefined commonly used curves according to the DRC Compression Curve Types listed in Table 7-13. If index *m_ucDRCCurveIndex* is equal to DRC_CURVE_TYPE_DTSUHD_COMMON_EXPLICIT_PARAMS additional parameters will define the piece-wise linear DRC curve explicitly according to Figure 7-1.

Table 7-13: DRC Compression Curve Types

m_ucDRCCurveIndex	DRC Compression Curve Type
0	DRC_CURVE_TYPE_NO_COMPRESSION
1	DRC_CURVE_TYPE_DTS_LEGACY_FILM_STANDARD
2	DRC_CURVE_TYPE_DTS_LEGACY_FILM_LIGHT,
3	DRC_CURVE_TYPE_DTS_LEGACY_MUSIC_STANDARD,
4	DRC_CURVE_TYPE_DTS_LEGACY_MUSIC_LIGHT,
5	DRC_CURVE_TYPE_DTS_LEGACY_SPEECH,
6	DRC_CURVE_TYPE_DTSUHD_COMMON_1,
7	DRC_CURVE_TYPE_DTSUHD_COMMON_2,
8	DRC_CURVE_TYPE_DTSUHD_COMMON_3,
9	DRC_CURVE_TYPE_DTSUHD_COMMON_4,
10	DRC_CURVE_TYPE_DTSUHD_COMMON_5,
11	DRC_CURVE_TYPE_DTSUHD_COMMON_6,
12	DRC_CURVE_TYPE_DTSUHD_COMMON_7,
13	DRC_CURVE_TYPE_DTSUHD_COMMON_8,
14	DRC_CURVE_TYPE_DTSUHD_COMMON_9,
15	DRC_CURVE_TYPE_DTSUHD_COMMON_EXPLICIT_PARAMS

Pseudo-code for extraction of parameters that describe the custom DRC curve can be found in Table 7-14.

Table 7-14: ExtractCustomDRCCurves

Syntax	Reference

`// Function that extracts custom DRC parameters`
`void CMFDStaticMD::ExtractCustomDRCCurves(CDRCProfile *pDRCProfile)`
`{`	

`  // Index into DRC_CURVE_TYPES table`	
`  pDRCProfile->m_ucDRCCurveIndex = (uchar)m_pStaticMDBitStream->ExtractBits(4);`	7.7.6.9.1
`  if (pDRCProfile->m_ucDRCCurveIndex == DRC_CURVE_TYPE_DTSX_E2_COMMON_EXPLICIT_PARAMS)`	
`  {`	
`    uint unDRCCurveCode = (uint) m_pStaticMDBitStream->ExtractBits(15);`	7.7.6.9.2
`    pDRCProfile->GetDRCCurveParamIndices(unDRCCurveCode);`	7.7.6.9.3
`    pDRCProfile->LookUpDRCCurveParameters();`	
`  }`	
`}`	

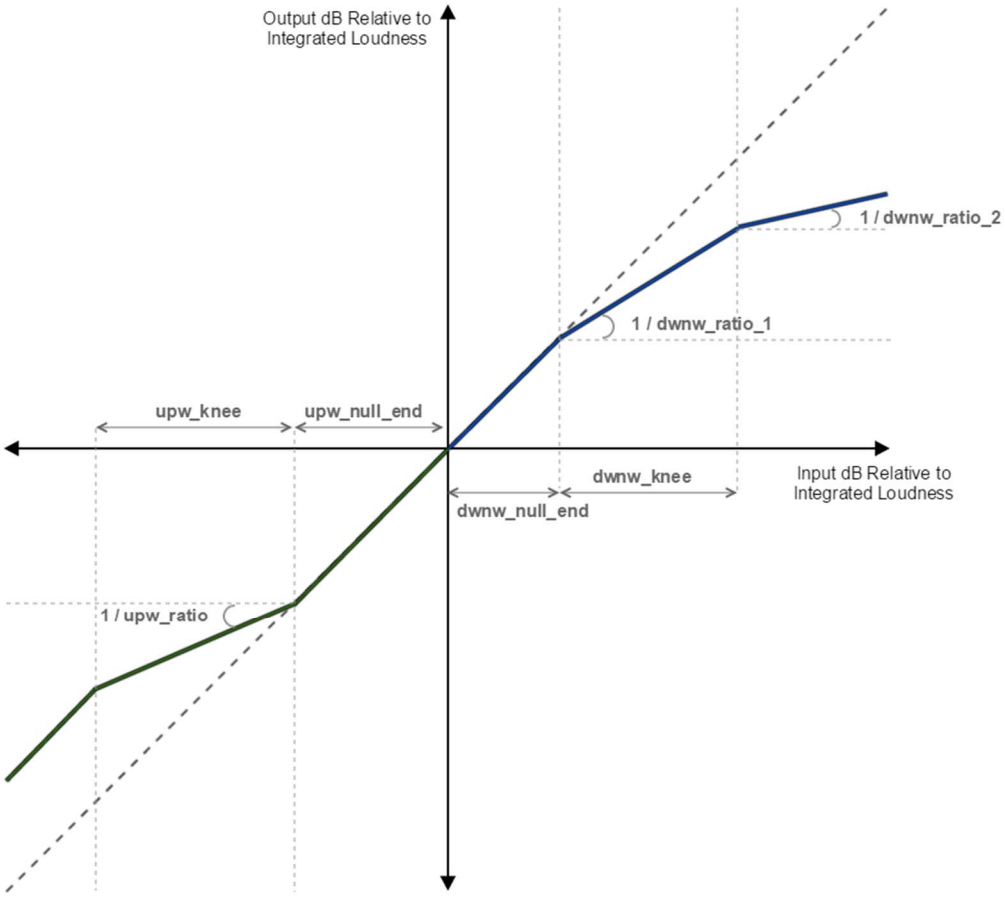

Figure 7-1: Piece-wise Linear DRC Curve

7.7.6.9 Parameters for ExtractCustomDRCCurves

7.7.6.9.1 m_ucDRCCurveIndex

This 4-bit field represents a look-up index into a table of DRC curve types according to the DRC Compression Curve Types.

7.7.6.9.2 unDRCCurveCode

This 15-bit field is present in the stream only if the *m_ucDRCCurveIndex* is equal to DRC_CURVE_TYPE_DTSUHD_COMMON_EXPLICIT_PARAMS. If the *unDRCCurveCode* is present, it represents a block code word that is used to encode all DRC curve parameters ordered as:

1) k=0: *m_rDrcDwnwCurveNullEnd* (denoted as *dwnw_null_end* in Figure 7-1)
2) k=1*: m_rDrcDwnwCurveRatio1* (denoted as *dwnw_ratio_1* in Figure 7-1)
3) k=2: *m_rDrcDwnwCurveRatio2* (denoted as *dwnw_ratio_2* in Figure 7-1)
4) k=3: *m_rDrcDwnwCurveKnee* (denoted as *dwnw_knee* in Figure 7-1)
5) k=4: *m_rDrcUpwCurveNullEnd* (denoted as *upw_null_end* in Figure 7-1)
6) k=5: *m_rDrcUpwCurveRatio1* (denoted as *upw_ratio* in Figure 7-1)
7) k=6: *m_rDrcUpwCurveKnee* (denoted as *upw_knee* in Figure 7-1).

Each parameter has an associated alphabet with allowed parameter values as listed in the respective table for each parameter (shown in Table 7-15). A value of a parameter *k* has an associated index within the corresponding k^{th} alphabet table denoted as *m_ucDRCParamIndices[k]*. A weight vector (*m_unDRCCurveCodeBasisWeights[]*) is created according to the sizes of parameter alphabets (*ucDRCCurveParamAlphabetSizes[]*) as follows:

- If the *ucDRCCurveParamAlphabetSizes*[7]={ 7, 3, 4, 3, 5, 3, 6 } then the corresponding weight vector *m_unDRCCurveCodeBasisWeights[7]* = {6*3*5*3*4*3, 6*3*5*3*4, 6*3*5*3, 6*3*5, 6*3, 6, 1}.
- A block code word u*nDRCCurveCode* is constructed from 7 parameters as:

$$unDRCCurveCode = \sum_{k=0..6}(m_ucDRCParamIndices[k] * m_unDRCCurveCodeBasisWeights[k])$$

  where *m_ucDRCParamIndices*[*k*] takes values from 0 to *m_unDRCCurveCodeBasisWeights*[*k*] - 1.

On the decoder side, as shown in pseudo-code within Table 7-15:

- the parameter indices *m_ucDRCParamIndices*[] are extracted from the received block code word *unDRCCurveCode* by means of iterative integer division and reminder calculation;
- the actual parameter values are obtained by table look-up.

7.7.6.9.3 Get DRC Curve

GetDRCCurveParamIndices() is a function to extract the DRC_ALPHABET_SIZE_TABLE_LENGTH indices from a single block code word given in *unDRCCurveCode*.

The weight associated with each index is given in *m_unDRCCurveCodeBasisWeights[]*. Block code is assumed to be generated according to:

$$unDRCCurveCode = \sum_{k=0..N}(m_ucDRCParamIndices[k] * m_unDRCCurveCodeBasisWeights[k])$$

where *N* = DRC_ALPHABET_SIZE_TABLE_LENGTH - 1.

The extracted indices are stored in the *m_ucDRCParamIndices*[].

Table 7-15: LookUpDRCCurveParameters

Syntax

```
#define DRC_ALPHABET_SIZE_TABLE_LENGTH 7    // Number of parameters that characterize a DRC curve

// Sizes of alphabet for each of the DRC curve parameters
// {Down Curve : Null, Ratio 1, Ratio 2, Knee, Up Curve : Null, Ratio 1, Knee }
const uchar ucDRCCurveParamAlphabetSizes[DRC_ALPHABET_SIZE_TABLE_LENGTH] = { 7, 3, 4, 3, 5, 3, 6
};

// Downward Compression Curve Parameter Alphabets
const float rDrcDwnwCurveNullEnd_Table[7] = { 0.0f, 2.5f, 5.0f, 10.0f, 15.0f, 20.0f, 25.0f };
const float rDrcDwnwCurveRatio1_Table[3] = { 2.0f, 4.0f, 20.0f };
const float rDrcDwnwCurveRatio2_Table[4] = { 0.0f, 4.0f, 8.0f, 20.0f };
const float rDrcDwnwCurveKnee_Table[3] = { 0.0f, 5.0f, 10.0f };

// Upward Compression Curve Parameter Alphabets
const float rDrcUpwCurveNullEnd_Table[5] = { 0.0f, -2.5f, -5.0f, -10.0f, -20.0f };
const float rDrcUpwCurveRatio1_Table[3] = { 2.0f, 4.0f, 8.0f };
const float rDrcUpwCurveKnee_Table[6] = { -5.0f, -10.0f, -15.0f, -20.0f, -25.0f, -30.0f };

{
    for (uint k = 0; k < DRC_ALPHABET_SIZE_TABLE_LENGTH; k++)
    {
        // Get k-th parameter index
        m_ucDRCParamIndices[k] = unDRCCurveCode / m_unDRCCurveCodeBasisWeights[k];
        // Calculate reminder for the next iteration
        unDRCCurveCode = unDRCCurveCode - m_ucDRCParamIndices[k] *
m_unDRCCurveCodeBasisWeights[k];
    }
}

// Function to look-up DRC curve parameters based on extracted indices in m_ucDRCParamIndices[]
// and an associated table for each parameter
void LookupDRCCurveParameters(void)
{
```

Syntax

```
    uint ni = 0;

    // Parameters Defining Downward Compression Curve
    //
    m_rDrcDwnwCurveNullEnd = rDrcDwnwCurveNullEnd_Table[ m_ucDRCParamIndices[ni++] ];
    m_rDrcDwnwCurveRatio1 = rDrcDwnwCurveRatio1_Table[ m_ucDRCParamIndices[ni++] ];
    m_rDrcDwnwCurveRatio2 = rDrcDwnwCurveRatio2_Table[ m_ucDRCParamIndices[ni++] ];
    m_rDrcDwnwCurveKnee = rDrcDwnwCurveKnee_Table[ m_ucDRCParamIndices[ni++] ];

    // Parameters Defining Upward Compression Curve
    //
    m_rDrcUpwCurveNullEnd = rDrcUpwCurveNullEnd_Table[ m_ucDRCParamIndices[ni++] ];
    m_rDrcUpwCurveRatio1 = rDrcUpwCurveRatio1_Table[ m_ucDRCParamIndices[ni++] ];
    m_rDrcUpwCurveKnee = rDrcUpwCurveKnee_Table[ m_ucDRCParamIndices[ni] ];
}
```

7.7.6.10 m_bCustomDRCSmoothMDPresent

A separate *m_bCustomDRCSmoothMDPresent* flag is sent for every DRC compression type found in Table 7-12.

If *m_bCustomDRCSmoothMDPresent* is TRUE, then for a particular DRC compression type the custom DRC smoothing parameters are transmitted in the stream. Otherwise, the decoder's default smoothing parameters are used for the particular DRC compression type.

7.7.6.11 m_rFastAttack

This 6-bit field is transmitted in the stream only if *bCustomDRCCurveMDPresent* is TRUE. It represents the smoothing time constant for the fast attack smoothing of the DRC compression.

The 6-bit index is mapped to 64 values in the range [0 to 25] using the *Quantization Table for DRC Fast Attack Smoothing Constant* in clause 5.2.2.4.

7.7.6.12 m_rSlowAttack

This 6-bit field is transmitted in the stream only if *bCustomDRCCurveMDPresent* is TRUE. It represents the smoothing time constant for the slow attack smoothing of DRC compression.

The 6-bit index (ind) is mapped to 64 values in the range [0 to 250] using the *Quantization Table for DRC Fast Attack Smoothing Constant* in clause 5.2.2.4 and multiplying the resulting indexed value by 10.

7.7.6.13 m_rFastRelease

This 6-bit field is transmitted in the stream only if *bCustomDRCCurveMDPresent* is TRUE. It represents the smoothing time constant for the fast release of DRC compression.

The 6-bit index is mapped to 64 values in the range [0 to 1 000] using the *Quantization Table for DRC Fast Release Smoothing Constant* as shown clause 5.2.2.5.

7.7.6.14 m_rSlowRelease

This 6-bit field is transmitted in the stream only if *bCustomDRCCurveMDPresent* is TRUE. It represents the smoothing time constant for the slow release of DRC compression.

The 6-bit index is mapped to 64 values in the range [0 to 1 000] using the *Quantization Table for DRC Fast Release Smoothing Constant* as shown clause 5.2.2.5, then multiply the resulting indexed value by 2.

7.7.6.15 m_rAttackThreshld

This 6-bit field is transmitted in the stream only if *bCustomDRCCurveMDPresent* is TRUE. The *m_rAttackThreshld* defines when to switch from a slow to a fast attack during DRC compression.

The 6-bit index is mapped to 64 values, in power domain in range [1 to 1 000]. The mapping is obtained by taking an inverse of the value extracted from the *Quantization Table for DRC Slow to Fast Threshold* shown in clause 5.2.2.6.

7.7.6.16 m_rReleaseThreshld

This 6-bit field is transmitted in the stream only if *bCustomDRCCurveMDPresent* is TRUE. The *m_rReleaseThreshld* defines when to switch from a slow to a fast release during DRC compression.

The 6-bit index is mapped to 64 values, in power domain in range [1,0 to 0,001], using the *Quantization Table for DRC Slow to Fast Threshold* shown in clause 5.2.2.6.

7.7.6.17 Reserved and Byte Alignment Fields

Since the overall size of the static metadata in bits is given by the *m_unNumStaticMDPackets*m_unStaticMDPacketByteSize*8, the new metadata may be added into the reserved fields without breaking the backward compatibility.

Decoders shall always navigate to the end of the static metadata block in order to continue with stream unpacking.

7.8 Extraction of Object Metadata

7.8.1 Overview of Object Metadata

Metadata for Objects includes an extensive set of parameters that describe the mixing environment, object location and advanced object properties such as dispersion and distance. Multiple sets of object metadata may be present for supporting different listening environments through the rendering exception parameters.

Two renderer models - '*basic*' and '*extended*' - are supported. The basic renderer model is intended to facilitate production of low cost and low power devices for use in non-ideal listening conditions. A basic renderer will treat all objects as a point source in space, where an extended renderer will support dispersion properties and the ability to characterize object position relative to the video display size and position.

The studio metadata, extracted in Table 7-16 of clause 7.8.2, determines whether an object is within the field of view of the video or outside the field of view. This metadata is only supported in the extended renderer model.

Some metadata parameters extracted in Table 7-30 of clause 7.8.11.15 and Table 7-34 of clause 7.8.11.27 characterizing dispersion angles and rotation of the object sources are also only utilized by the extended renderer model.

All other metadata parameters are processed identically by both the basic renderer model and the extended renderer model.

7.8.2 ExtractMetadataForObjects

Pseudo-code that describes the extraction of this block of metadata is shown in Table 7-16.

Table 7-16: ExtractMetadataForObjects

Syntax	Reference

`void C0x01MDChunkBody::ExtractMetadataForObjects(bool bSyncFrameFlag,`
`bool bFullChannelBasedMixFlag,`
`uint unFrameDuration,`
`uint unOutChMask,`
`bool bInteractObjLimitsPresent,`
`std::vector<uint> & punObjectIDList)`
`{`
`  uint unObjectID;`
`  uint unObjIndex;`
`  bool bObjStartFrameFlag;  // Flag indicating if this is the first frame of`
`// object appearance in the interval between the two sync frames`
 	
`  if (bSyncFrameFlag) {`
`  // In sync frame reset the object list`
`  m_CObjList.ResetList();`	

Syntax	Reference

}
else {
m_CObjList.ResetObjMDPresentFlags();
}
// Mixing studio environment parameters
if (bSyncFrameFlag && (bFullChannelBasedMixFlag==FALSE)) {
m_bMixStudioParamsPresent = (bool)ExtractBits(1);
	7.8.3.1

if (m_bMixStudioParamsPresent) { m_ucRadiusRefernceUnitSphereInMeters = (uchar) ExtractBits(4) + 3;	7.8.3.2
m_unRefScreenHorizontalViewingAngleInDeg = 10 * ((uint)ExtractBits(4)+ 1) + 20;	7.8.3.3

m_ucRefScreenAspectRatio = (uchar) ExtractBits(3);
} // End of if (bMixStudioParamsPresent)
}
for (uint nso=0; nso<m_unNumObjects; nso++) {
unObjectID = punObjectIDList[nso];
	7.8.3.4

bool bMetadatSuitableforREFlag = CheckIfMDIsSuitableforImplObjRenderer(unObjectID);	7.8.4

if (bMetadatSuitableforREFlag) {
unObjIndex = m_CObjList.CheckIfNewObject(unObjectID);
if (m_CObjList.m_pArrayOfObjPntrs[unObjIndex] == NULL) {
bObjStartFrameFlag = TRUE; // Indicates appearance of the new object
// Allocate new object to the list
m_CObjList.m_punObjIDs[unObjIndex] = unObjectID; // Save the Object ID
m_CObjList.m_pArrayOfObjPntrs[unObjIndex] = new CLObjMD(m_pBitStream);
// Save the object ID
m_CObjList.m_pArrayOfObjPntrs[unObjIndex]->m_unObjectID = unObjectID;
}
else {
bObjStartFrameFlag = FALSE;
}
m_CObjList.m_pbObjMDPresent[unObjIndex] = TRUE;
// Extract object static flag (present only when bObjStartFrameFlag is TRUE)
if (bObjStartFrameFlag == TRUE) {
if (unObjectID == 256) {
m_CObjList.m_pArrayOfObjPntrs[unObjIndex]->m_bObjStaticFlag = TRUE;
}
else { // Case for unObjectID != 256
m_CObjList.m_pArrayOfObjPntrs[unObjIndex]->m_bObjStaticFlag =
	7.8.6

(bool)ExtractBits(1);
}
}
// Extract metadata for either objects or object group definitions
m_CObjList.m_pArrayOfObjPntrs[unObjIndex]->ExtractObjectsOrObjGroupDef(
bObjStartFrameFlag,
unFrameDuration,
unOutChMask,
bInteractObjLimitsPresent);
	7.8.7

} // End of if (bMetadatSuitableforREFlag)
} // End of for (uint nso=0; nso<m_unNumObjects; nso++)
// Delete De-register objects that have disappeared in this frame.
m_CObjList.PurgeList();
} // End of C0x01MDChunkBody::ExtractMetadataForObjects()	

7.8.3 Parameters for ExtractMetadataForObjects

7.8.3.1 m_bMixStudioParamsPresent

This 1-bit flag is present in the stream only if *bSyncFrameFlag* is TRUE and *bFullChannelBasedMixFlag* is FALSE. When *m_bMixStudioParamsPresent* is TRUE, the parameters that describe the reference monitoring environment are present.

These parameters are present so the rendering engine can modify object position in order to reflect the difference between the screen size in the reference monitoring environment and the screen size in the listening environment.

7.8.3.2 m_ucRadiusRefernceUnitSphereInMeters

This 4-bit field is present in the stream only if *m_bMixStudioParamsPresent* is TRUE. The value of *m_ucRadiusRefernceUnitSphereInMeters* indicates the radius of a reference unit sphere in meters (i.e. distance to the center speaker or distance to the screen in the reference monitoring environment).

Valid range is from 3 meters to 18 meters with step size of 1 meter. If not transmitted the reference radius is by default set to *m_ucRadiusRefernceUnitSphereInMeters* = 8 meters.

7.8.3.3 m_unRefScreenHorizontalViewingAngleInDeg

This 4-bit field is present in the stream only if *m_bMixStudioParamsPresent* is TRUE. The value of *m_unRefScreenHorizontalViewingAngleInDeg* represents the horizontal viewing angle subtended by the screen relative to an observer located at the optimal viewing position, in degrees.

Valid values are in range from 30 to 180 degrees in steps of 10 degrees. If not transmitted the *m_unRefScreenHorizontalViewingAngleInDeg* is by default set to 50 degrees.

7.8.3.4 m_ucRefScreenAspectRatio

This 3-bit field is present in the stream only if the *m_bMixStudioParamsPresent* is TRUE. The value of *m_ucRefScreenAspectRatio* represents a look-up index into Table 7-17.

The default value for *m_ucRefScreenAspectRatio* is REFERENCE_SCREEN_ASPECT_RATIO_SCOPE.

Table 7-17: Reference Screen Aspect Ratios

Index	Mnemonic	Description
0	REFERENCE_SCREEN_ASPECT_RATIO_FLAT	Flat: aspect ratio close to 1.85:1
1	REFERENCE_SCREEN_ASPECT_RATIO_SCOPE	SCOPE: aspect ratio close to 2.39:1
2	REFERENCE_SCREEN_ASPECT_RATIO_HDTV	HDTV: aspect ratio close to 1.78:1
3	REFERENCE_SCREEN_ASPECT_RATIO_FULL	Full aspect ratio close to1.9:1
4	REFERENCE_SCREEN_ASPECT_RATIO_RESERVED1	Reserved for future use
5	REFERENCE_SCREEN_ASPECT_RATIO_RESERVED2	Reserved for future use
6	REFERENCE_SCREEN_ASPECT_RATIO_RESERVED3	Reserved for future use
7	REFERENCE_SCREEN_ASPECT_RATIO_RESERVED4	Reserved for future use

7.8.4 Association of 3D Object Metadata to 3D Renderer Type

This metadata block code extracts renderer association flags and checks if this metadata is suitable for the current renderer implementation. If metadata is not suitable, the bitstream read pointer is advanced to the metadata for the next object. Pseudo-code can be found in Table 7-18.

Table 7-18: CheckIfMDIsSuitableforImplObjRenderer

Syntax	Reference
`bool C0x01MDChunkBody::CheckIfMDIsSuitableforImplObjRenderer(uint unObjectID)`	
`{`	
`  bool bMetadatSuitableforREFlag;`	
`  if (unObjectID >= OBJGROUPIDSTART)`	
`  {`	
`    // Object groups are by default suitable for all renderer implementations`	
`    bMetadatSuitableforREFlag = TRUE;`	
`  }`	
`  else`	
`  {`	
`    bool bMDUsedByAllRenderersFlag = (bool)ExtractBits(1);`	7.8.5.1
`    if (bMDUsedByAllRenderersFlag)`	
`    {`	
`      // Metadata suitable for all renderer implementations`	
`      bMetadatSuitableforREFlag = TRUE;`	
`    }`	
`    else`	
`    {`	
`      uchar ucRequiredRendererType = (uchar) ExtractBits(1);`	7.8.5.2

Syntax	Reference

uchar uctable[4] = { 8, 10, 12, 14 };
uint unNumBits2Skip = (uint) ExtractVarLenBitFields(uctable) + 1;
if (ucRequiredRendererType == IMPLEMENTED_OBJECT_RENDERER_TYPE) {
bMetadatSuitableforREFlag = TRUE;
}
else {
bMetadatSuitableforREFlag = FALSE;
AdvanceBitReadPointer(unNumBits2Skip);
}
} // End of case when the bMDUsedByAllRenderersFlag is FALSE
} // End of case when (unObjectID<OBJGROUPIDSTART)
return bMetadatSuitableforREFlag;
} // End of CheckIfMDIsSuitableforImplObjRenderer()
	7.8.5.3

7.8.5 Parameters for CheckIfMDIsSuitableforImplObjRenderer

7.8.5.1 bMDUsedByAllRenderersFlag

This 1-bit flag is transmitted only if the *unObjectID* < OBJGROUPIDSTART i.e. it is not transmitted for object groups. Its value indicates if object metadata is applicable to:

- both the basic 3D object renderer type and the extended 3D renderer type (when *bMDUsedByAllRenderersFlag* is TRUE); or
- just one of the 3D object renderer types (when *bMDUsedByAllRenderersFlag* is FALSE).

If the *bMDUsedByAllRenderersFlag* is TRUE, regardless of the implemented renderer type (IMPLEMENTED_OBJECT_RENDERER_TYPE), the object ID shall be registered and the corresponding metadata shall be extracted.

7.8.5.2 ucRequiredRendererType

This 1-bit field is interpreted according to Table 7-19 and is transmitted only if *unObjectID* < OBJGROUPIDSTART and *bMDUsedByAllRenderersFlag* are FALSE. Its value indicates the association of the 3D object metadata to a particular renderer type.

If *bMDUsedByAllRenderersFlag* is FALSE, nominally there will be two sets of metadata with the same unObjectID:

- one for the basic renderer (when *ucRequiredRendererType* = OBJECT_RENDERER_TYPE_BASIC); and
- one for the extended renderer (when *ucRequiredRendererType* = OBJECT_RENDERER_TYPE_EXTENDED).

The metadata is used only if *ucRequiredRendererType* is less than or equal to the implemented renderer type as indicated by the IMPLEMENTED_OBJECT_RENDERER_TYPE.

If *bMDUsedByAllRenderersFlag* is TRUE, then *ucRequiredRendererType* is by default set to OBJECT_RENDERER_TYPE_BASIC.

Table 7-19: ucRequiredRendererType

value	ucRequiredRendererType
0	OBJECT_RENDERER_TYPE_BASIC
1	OBJECT_RENDERER_TYPE_EXTENDED

7.8.5.3 unNumBits2Skip

This variable length field is transmitted only if the *unObjectID* < OBJGROUPIDSTART and *bMDUsedByAllRenderersFlag* is FALSE. Its value indicates the size, in bits, of the remaining part of this object metadata. The variable length codes of 9, 12, 15 or 17 bits are used to allow the *unNumBits2Skip* values up to 256, 1 280, 5 376 or 21 760 bits.

If 3D object metadata is not suitable for the implemented object renderer (*ucRequiredRendererType* is greater than the IMPLEMENTED_OBJECT_RENDERER_TYPE) then the *unNumBits2Skip* is used to navigate to the next object's metadata.

7.8.6 m_bObjStaticFlag

This flag indicates if the object properties are static (not changing) until the appearance of the next sync frame. This flag is transmitted only in the frames in which the object appears for the first time within the period between the two consecutive sync frames (*bObjStartFrameFlag* is TRUE).

7.8.7 Extraction of 3D Object/Object Group Metadata

An object ID (*m_unObjectID*) indicates whether the metadata to be extracted is:

- metadata for an object (*m_unObjectID* = 0, 1, ... OBJGROUPIDSTART-1), describing 3D object properties and directly points to particular waveforms encoded within an audio chunk; or
- metadata for an object group (*m_unObjectID* = OBJGROUPIDSTART, ... 255) which lists object IDs belonging to the group and does not directly point to any audio waveforms; or
- metadata for full channel mask-based mix (*m_unObjectID* = 256) describing the mix and directly points to particular waveforms encoded within an audio chunk.

The value of OBJGROUPIDSTART is set to 224.

An example for the use of object groups and 3D objects is illustrated in Figure 7-2 below. Boxes in purple color indicate the object groups and the objects that are active by default. A particular choice is defined by values of:

- *m_bObjGrpActiveFlag* and *m_unWithinGrpObjActMask* in each of the object groups; and
- *m_bObjActiveFlag* in Object 5.

Values of *m_bObjActiveFlag* for Objects 1 to 4 are irrelevant since they are overruled by the corresponding bits within the *unWithinGrpObjActMask.*

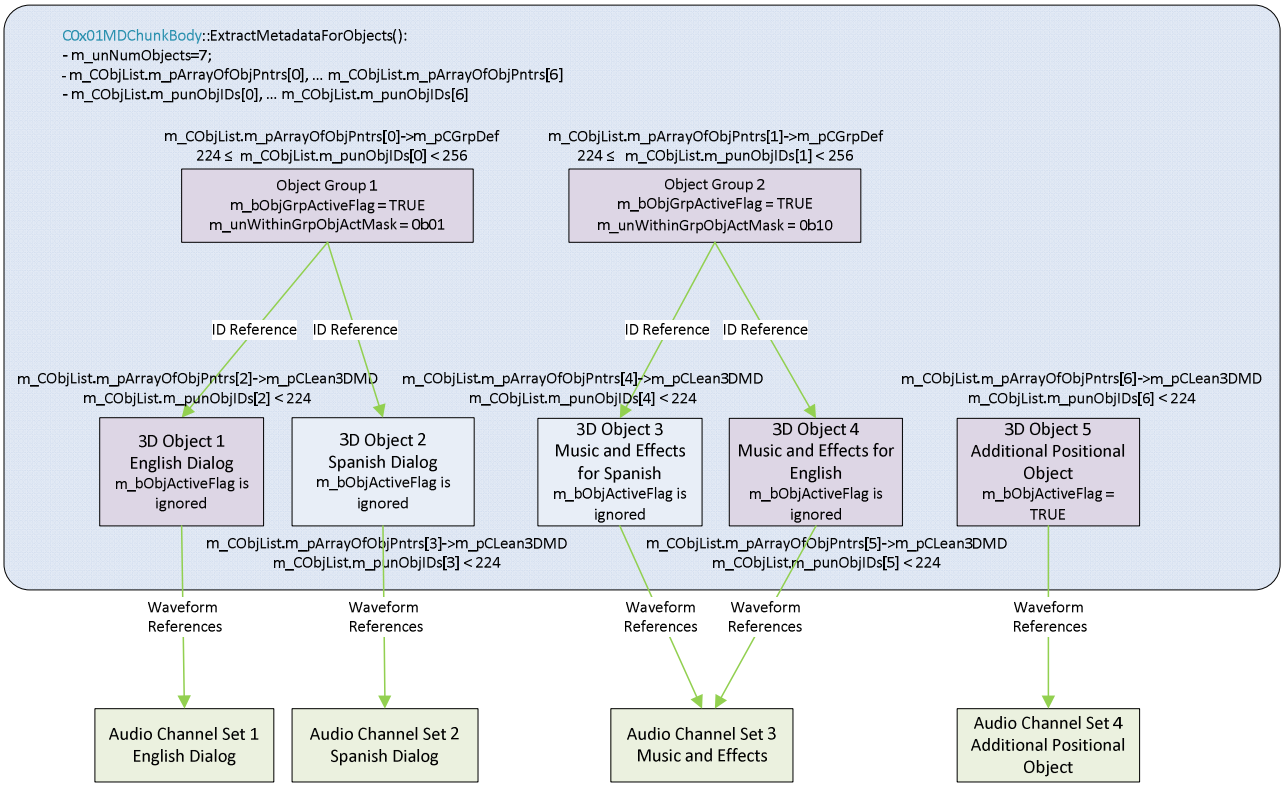

Figure 7-2: Example of Object Groups and 3D Objects

Note that both 3D Object 3 and 3D Object 4 point to the same audio waveforms hence the only difference between these two objects is in the associated metadata (i.e. dynamic object gain dependent on the associated speech object).

For static 3D objects/object groups the associated metadata is transmitted only once during the interval between the two sync frames. Static objects cannot disappear before the next sync frame hence they are persistent until the next sync frame. The decoder shall inherit all of the static object metadata from the corresponding object metadata used in the previous frame.

For static group objects the group definition cannot change until the next sync frame. 3D objects belonging to a static group object do not have to be static by themselves (i.e. the changes of metadata fields within associated 3D objects are allowed). However, 3D objects belonging to a static group are persistent as long as the group is static.

Pseudo-code for extraction of this block of metadata can be found in Table 7-20.

Table 7-20: ExtractObjectsOrObjGroupDef

Syntax	Reference

`void CLObjMD::ExtractObjectsOrObjGroupDef(bool bObjStartFrameFlag,`
`  uint unFrameDuration,`
`  uint unOutChMask,`
`  bool bInteractiveObjectsLimitsPresent)`
`{`
`  if (m_unObjectID >= OBJGROUPIDSTART && m_unObjectID<256) {`
`  if (m_pCGrpDef != NULL) {`
`  m_pCGrpDef = new CLGrpDef(m_pBitStream);`
`  }`
`  m_pCGrpDef->m_bObjGrpStaticFlag = m_bObjStaticFlag;`
`  m_pCGrpDef->ExtractGroupDefinition(bObjStartFrameFlag);`
`  }`
`  else {`
`  // Case of Object IDs 0 to OBJGROUPIDSTART-1 and 256.`
`  if (m_pCLean3DMD != NULL) {`
`  m_pCLean3DMD = new CL3DMD(m_pBitStream);`
`  }`
`  m_pCLean3DMD->m_bObjStaticFlag = m_bObjStaticFlag;`
`  // Object IDs 0 to OBJGROUPIDSTART-1 and 256 are reserved for objects.`
`  m_pCLean3DMD->ExtractObjectMetadata(bInteractiveObjectsLimitsPresent,`
`  bObjStartFrameFlag,`
`  m_unObjectID,`
`  unFrameDuration,`
`  unOutChMask);`
`  }`
`} // End of the ExtractObjectsOrObjGroupDef() function`
	7.8.8
	7.8.10

7.8.8 Object Group Definition

Object group metadata does not carry any specific 3D object properties nor pointers to the audio waveforms. Instead, the object group metadata carries lists of object IDs belonging to the group and the indication of which of these objects shall be played by default. Both 3D objects and other object groups may be referenced within the object group metadata as long as they are all associated with (defined within) the same 0x01 metadata chunk and the group hierarchy follows the packing order (i.e. only the objects/groups packed prior to the current object group can be referenced within this group).

If group is defined as non-static (*bObjGrpStaticFlag* is FALSE) the group definition metadata may be transmitted in every frame. If group is static, the group definition metadata may be transmitted only in frames where the *bObjStartFrameFlag* is TRUE i.e. when *(m_bObjGrpStaticFlag* && *bObjStartFrameFlag*) is TRUE.

Pseudo-code for extraction of object group definition metadata can be found in Table 7-21.

Table 7-21: ExtractGroupDefinition

Syntax	Reference
`void CLGrpDef::ExtractGroupDefinition(bool bObjStartFrameFlag)`	
`{`	
`  bool bGrpObjRefUpdFlag;`	
`  if((m_bObjGrpStaticFlag==FALSE) || ((m_bObjGrpStaticFlag) && (bObjStartFrameFlag)))`	
`  {`	
`    m_bObjGrpActiveFlag = (ExtractBits(1) == 1) ? TRUE : FALSE;`	7.8.9.1
`    bGrpObjRefUpdFlag = (bObjStartFrameFlag) ? TRUE : (bool) ExtractBits(1);`	7.8.9.2
`    if (bGrpObjRefUpdFlag)`	
`    {`	
`      uchar uctable[4]={1, 3, 3, 4};`	
`      m_ucNumObjReferencedInGroup = (uchar) ExtractVarLenBitFields(uctable) + 1;`	7.8.9.3
`      for (uchar nid=0; nid<m_ucNumObjReferencedInGroup; nid++) {`	
`        uint unNumBitsforObjID = (ExtractBits(1)==1) ? 8 : 4;`	6.3.5.2
`        m_punObjGrpRefIDs[nid] = (uint) ExtractBits(unNumBitsforObjID);`	7.8.9.4
`      } // End of for (nid=0; nid m_ucNumObjReferencedInGroup`	
`    } // End of if (bGrpObjRefUpdFlag)`	
`    bool bUpdFlag = (bObjStartFrameFlag || bGrpObjRefUpdFlag) ? TRUE : (bool) ExtractBits(1);`	7.8.9.5
`    if (bUpdFlag) {`	
`      // Extract new mask (m_ucNumObjReferencedInGroup bits wide)`	
`      m_unWithinGrpObjActMask = ExtractBits(m_ucNumObjReferencedInGroup);`	7.8.9.6
`    }`	
`  (bObjStartFrameFlag==TRUE) ) )`	
`    // Object Group Metadata Extension (for objects with unObjectID>OBJGROUPIDSTART)`	
`    // If TRUE it indicates the presence of object metadata extension.`	
`    m_bObjGroupMDExtensionPresent = ExtractBits(1);`	7.8.9.7
`    if (m_bObjGroupMDExtensionPresent)`	
`    {`	
`      uchar uctable[4] = { 5, 7, 9, 11 };`	
`      // Values in uctable[] allow unTemp to be up to 32, 160, 672 or 2720 bits.`	
`      // Prefix-code of length 1, 2, 3 and 3 is carried correspondingly for`	
`      // each of the values in uctable.`	
`            uint unNumBitsToExtract = (uint)ExtractVarLenBitFields(uctable) + 1;`	7.8.9.8
`      AdvanceBitReadPointer(unNumBitsToExtract);`	7.8.9.9
`    }`	
`  } // End of if ( ( m_bObjGrpStaticFlag == FALSE ) || ( (m_bObjGrpStaticFlag) …`	
`} // End of CLGrpDef::ExtractGroupDefinition()`	

7.8.9 Parameters for ExtractGroupDefinition

7.8.9.1 m_bObjGrpActiveFlag

This 1-bit field indicates if an object group is active by default (*m_bObjActiveFlag* is TRUE), or not (*m_bObjActiveFlag* is FALSE).

If a system layer explicitly sets the object group activity, *m_bObjGrpActiveFlag* extracted within the stream shall be ignored.

7.8.9.2 bGrpObjRefUpdFlag

This 1-bit flag when TRUE, indicates that there is an update to either the number of objects or the list of object IDs that are referenced within this group. In the sync frames *bGrpObjRefUpdFlag* is not transmitted but is set to TRUE by default.

7.8.9.3 m_ucNumObjReferencedInGroup

This field is transmitted in the stream only if the *bGrpObjRefUpdFlag* is TRUE. The value of *m_ucNumObjReferencedInGroup* represents the number of objects that are referenced within the group. Variable length code of 2, 5, 6 and 7 bits is used to represent the values of *m_ucNumObjReferencedInGroup* up to 2, 10, 18 or 32 object references respectively.

7.8.9.4 m_punObjGrpRefIDs[]

These fields are transmitted in the stream only if *bGrpObjRefUpdFlag* is TRUE. The list of object IDs that are referenced within this group is stored in the array *m_punObjGrpRefIDs[]*. Each object ID is coded as a variable length code with length of either 5 or 9 bits allowing up to 16 or 256 object ID references respectively.

7.8.9.5 bUpdFlag

bUpdFlag is not present in the bitstream and shall be set to TRUE when *bObjStartFrameFlag* is TRUE, or *bGrpObjRefUpdFlag* is TRUE. Otherwise, it is read directly from the bitstream.

7.8.9.6 m_unWithinGrpObjActMask

m_unWithinGrpObjActMask is a *m_ucNumObjReferencedInGroup*-bit wide bit-mask that represents the group's default play list, i.e. the list of objects that shall be decoded and played by default. An application layer through decoder APIs can overwrite this default.

m_unWithinGrpObjActMask is updated when *bUpdFlag* is TRUE. If *bUpdFlag* is FALSE, then it maintains its value from the previous frame.

The *m_unWithinGrpObjActMask* is coded as a bit-mask where each bit corresponds to an object. Bits within the mask follow the order of object ID packing in *m_punObjGrpRefIDs* where the LSB -> first object, MSB -> last object.

7.8.9.7 m_bObjGroupMDExtensionPresent

When *m_bObjGroupMDExtensionPresent* is TRUE, an Object Group Metadata Extension Field is transmitted in the stream.

7.8.9.8 unNumBitsToExtract

Size of *Object Group Metadata Extension Fields*. This variable length field is transmitted in the stream if the *m_bObjGroupMDExtensionPresent* is TRUE. Variable code of length 5, 7, 9 or 11 bits allow the size of the extension fields to be up to 32, 160, 672 or 2 720 bits.

7.8.9.9 Object Group Metadata Extension Fields

This block of data is present in the stream if the *m_bObjGroupMDExtensionPresent* is TRUE. Decoders built to this version of the specification shall navigate past these metadata fields in order to parse the metadata for the next object.

7.8.10 3D Object Metadata

The underlying principle of 3D object metadata transmission is that only metadata that has changed value from the previous frame is retransmitted in the current frame. However, in order to facilitate random navigation, the full set of metadata is re-transmitted at least once within the period between the two consecutive sync frames.

In frames where *bObjStartFrameFlag* is TRUE, all active metadata is re-transmitted. In all consecutive frames with the *bObjStartFrameFlag*=FALSE, the metadata that is not present is assumed to keep the same value from the previous frame.

Pseudo-code for the extraction of 3D object metadata can be found in Table 7-22.

Table 7-22: ExtractObjectMetadata

Syntax	Reference

`void CL3DMD::ExtractObjectMetadata(bool bInteractiveObjectsLimitsPresent,`
`  bool bObjStartFrameFlag,`
`  uint unObjectID,`
`  uint unFrameDuration,`
`  uint unOutChMask)`
`{`
`  uint unTemp;`
`  m_bObjActiveFlag = (unObjectID == 256) ? TRUE : (bool)ExtractBits(1);`
`  if (bObjStartFrameFlag == TRUE) {`
	7.8.11.1

Syntax	Reference
m_ucObjRepresTypeIndex = ExtractBits(3);	7.8.11.2
m_bChMaskObjectFlag = false;	
m_bObject3DMetaDataPresent = false;	
m_bMonoObjWithMultipleSourcesFlag = false;	
m_bAmbisonicEncObjFlag = false;	
// set flags based on extracted representation type	
switch (m_ucObjRepresTypeIndex) {	
case REP_TYPE_CH_MASK_BASED:	
case REP_TYPE_MTRX2D_CH_MASK_BASED:	
case REP_TYPE_MTRX3D_CH_MASK_BASED:	
m_bChMaskObjectFlag = true;	
break;	
case REP_TYPE_3D_OBJECT_SINGLE_SRC_PER_WF:	
m_bObject3DMetaDataPresent = true;	
break;	
case REP_TYPE_MONO_3D_OBJECT_MULTI_SRC_PER_WF:	
m_bObject3DMetaDataPresent = true;	
m_bMonoObjWithMultipleSourcesFlag = true;	
break;	
case REP_TYPE_BINAURAL:	
m_bChMaskObjectFlag = true;	
break;	
case REP_TYPE_AUDIO_TRACKS:	
break;	
case REP_TYPE_AMBISONIC: // Ambisonic representation	
m_bAmbisonicEncObjFlag = true;	
break;	
default: // Error	
}	
if (unObjectID == 256) {	
m_ucObjectImportanceLevel = 7; // default to have the highest importance	
m_unObjTypeDescrIndex = ASSET_TYPE_COMPLETE_AUDIO_PRESENTATION;	
m_ucObjAudioChunkIndex = (uchar)0;	
m_ucObjNaviWitinACIndex = (uchar)0;	
} // End of case when the unObjectID==256	
else { // Case for unObjectID != 256	
m_ucObjectImportanceLevel = (uchar)ExtractBits(3);	7.8.11.3
// Object type description is obtained from the ASSET_TYPE	
bool bObjTypeDescrPresent = (ExtractBits(1) == 1) ? true : false;	7.8.11.4
if(bObjTypeDescrPresent){	
uchar ucTypeBitWidth = (ExtractBits(1)==1) ? 3 : 5;	7.8.11.5
m_unObjTypeDescrIndex = (uint) ExtractBits(ucTypeBitWidth);	
}	
else {	
// Default is an unknown type.	
m_unObjTypeDescrIndex = ASSET_TYPE_UNKNOWN;	
}	
// Pointers to Audio Waveforms Associated with this Object	
// Audio chunk index holding the object waveforms	
// Allows maximum index to be 1, 17, 33 or 255 respectively.	
// Prefix-code of length 1, 2, 3 and 3 is carried correspondingly for	
// each of the values in uctable.	
uchar uctable[4] = { 1, 4, 4, 8 };	
unTemp = (uint)ExtractVarLenBitFields(uctable);	
if (unTemp < 256){	
m_ucObjAudioChunkIndex = unTemp;	7.8.11.6
}	
uchar uctable[4] = { 3, 3, 4, 8 };	
unTemp = (uint)ExtractVarLenBitFields(uctable);	
if (unTemp < 256){	
m_ucObjNaviWitinACIndex = unTemp;	7.8.11.7
}	
m_bPerObjLTLoudnessMDPresent = (ExtractBits(1) == 1) ? true : false;	7.8.11.8
if (m_bPerObjLTLoudnessMDPresent){	
if (m_pObjLTLoudnessMD == NULL) {	
m_pObjLTLoudnessMD = new CObjLTLoudnessMD(m_pBitStream);	
}	
}	
// Extract per object long term loudness measurement parameters.	
m_pObjLTLoudnessMD->ExtractLTLoudnessMD();	7.8.11.9
} // End of case for unObjectID != 256	
// Extract object interactivity related metadata	
m_ObjInterLimits.ExtractObjectInteractMD(bInteractiveObjectsLimitsPresent, unObjectID, m_bObject3DMetaDataPresent);	7.8.11.11

Syntax	Reference
`// Initialize flags that may not be transmitted in all use cases`	
`m_bObjectRendExceptPresent = false;`	
`if (m_bChMaskObjectFlag == true) {`	
`  // Case when the m_bChMaskObjectFlag is true`	
`    // Metadata describing the channel mask based object.`	
`    ExtractChMaskParams();`	7.8.11.13
`} // End of m_bChMaskObjectFlag == true part`	
`else{ // if m_bChMaskObjectFlag == false`	
`  if (m_bObject3DMetaDataPresent){`	
`  // Case of 3D object metadata`	
`    if (m_bMonoObjWithMultipleSourcesFlag){`	
`      m_ucNumWaveFormsInObj = 1;`	
`    }`	
`    else{`	
`      // Number of waveforms defined for this object`	
`      // It allows for 2, 10, 18 or 32 waveforms in total.`	
`      // Prefix-code of length 1, 2, 3 and 3 is carried correspondingly for`	
`      // each of the values in uctable.`	
`      uchar uctable[4] = { 1, 3, 3, 4 };`	
`      m_ucNumWaveFormsInObj = (uchar)ExtractVarLenBitFields(uctable) + 1;`	7.8.11.17
`    }`	
`    // 3D Renderer configuration related metadata`	
`    ExtractRendererConfigParams();`	7.8.11.15
`  } // End of m_bObject3DMetaDataPresent = true part`	
`  else{`	
`    // Number of waveforms defined for this object`	
`    // It allows for 2, 10, 18 or 32 waveforms in total.`	
`    // Prefix-code of length 1, 2, 3 and 3 is carried correspondingly for`	
`    // each of the values in uctable.`	
`    uchar uctable[4] = { 1, 3, 3, 4 };`	
`    m_ucNumWaveFormsInObj = (uchar)ExtractVarLenBitFields(uctable) + 1;`	7.8.11.17
`    // When m_bAmbisonicEncObjFlag is true the encoded waveforms represent`	
`    // Ambisonics encoded signals (spherical harmonics).`	
`    if (m_bAmbisonicEncObjFlag){`	
`      // Extract Encoded Ambisonics Format Description`	
`      if (m_pAmbiReprMD == NULL){`	
`        m_pAmbiReprMD = new CAmbisonicReprMD(m_pBitStream);`	
`      }`	
`      m_pAmbiReprMD->ExtractAmbisonicsMD(m_ucNumWaveFormsInObj);`	7.8.11.18
`      m_bObjectRendExceptPresent = (ExtractBits(1) == 1) ? true : false;`	7.8.11.20
`    } // End of m_bAmbisonicEncObjFlag=true case`	
`    else{`	
`      m_bObjectRendExceptPresent = (ExtractBits(1) == 1) ? true : false;`	7.8.11.20
`    } // End of m_bAmbisonicEncObjFlag=false case`	
`  } // End of m_bObject3DMetaDataPresent = false part`	
`} // end of if (m_bChMaskObjectFlag == false)`	
`} // End of the case when bObjStartFrameFlag==true`	
`  // Metadata Possibly Present in all frames multiple times`	
`ExtractInvQuantMultiUpdtObjMD(bObjStartFrameFlag, unObjectID, unFrameDuration, unOutChMask);`	7.8.11.21
`if (m_bObject3DMetaDataPresent || m_bObjectRendExceptPresent) {`	
`// Exponential window lambda parameter characterizing the smoothing of this object's`	
`// contribution coefficients into the output bus.`	
`// If present the 5-bit index is transmitted in the stream.`	
`  unTemp = m_rPerObjExpWinLambda.UpdateCode(bObjStartFrameFlag, 5);`	7.8.11.23
`  if (m_rPerObjExpWinLambda.m_bUpdateFlag){`	
`    // Inverse quantization by table look-up`	
`    dtsflp rTemp = m_rExpWinLambdaTable[unTemp];`	
`    // Set the new parameter value`	
`    m_rPerObjExpWinLambda.SetParameter(rTemp);`	
`  }`	
`}`	
`// Object Metadata Extension (for objects with unObjectID<256)`	
`// If true it indicates the presence of object metadata extension.`	
`m_bObjecMDExtensionPresent = (unObjectID < 256) ? (bool)ExtractBits(1) : false;`	7.8.11.24
`if (m_bObjecMDExtensionPresent){`	
`  uchar uctable[4] = { 7, 9, 11, 13 };`	
`  // Values in uctable[] allow unTemp to be up to 128, 640, 2688 or 10880 bits.`	
`  // Prefix-code of length 1, 2, 3 and 3 is carried correspondingly for`	
`  // each of the values in uctable.`	
`  unTemp = (uint)ExtractVarLenBitFields(uctable) + 1;`	7.8.11.25
`  // Currently this metadata fields are not defined and decoders shall:`	
`  // 1)   extract block of data of length unNumBitsToExtract (in bits)`	
`  // 2)   discard the extracted data`	
`  // 3)   move on to the next object metadata`	
`  AdvanceBitReadPointer(unTemp);`	7.8.11.26

Syntax	Reference
} } // End of CL3DMD::void ExtractObjectMetadata(	

7.8.11 Parameters for ExtractObjectMetadata

7.8.11.1 m_bObjActiveFlag

This 1-bit field indicates if an object on its own, without being referenced in any object group, shall be rendered (*m_bObjActiveFlag* is TRUE), or not (*m_bObjActiveFlag* is FALSE).

If the object is referenced in any of the object groups then the group's object activity mask (*unInGrpObjActMask*) or a system layer object activity request shall be used to decide whether this object shall be rendered or not and the *m_bObjActiveFlag* shall be ignored.

For full channel mask-based presentation (*unObjectID*==256) the *m_bObjActiveFlag* is TRUE by default.

7.8.11.2 m_ucObjRepresTypeIndex

This 3-bit field represents an index into a table of object representation types as given in Table 7-23. The valid range of *m_ucObjRepresTypeIndex* is 0 to 7 when *unObjectID*<OBJGROUPIDSTART or 0 to 5 when *unObjectID* equal to 256.

There are four flags that control the further extraction of object metadata. All four flags are initially set to FALSE. Based on the value of *m_ucObjRepresTypeIndex* the state of these flags is modified accordingly. In particular:

- *m_bChMaskObjectFlag* indicates if the representation is channel mask-based; it is set to TRUE if the *m_ucObjRepresTypeIndex* is one of:
  - REP_TYPE_CH_MASK_BASED;
  - REP_TYPE_MTRX2D_CH_MASK_BASED; or
  - REP_TYPE_MTRX3D_CH_MASK_BASED; or
  - REP_TYPE_BINAURAL.
- *m_bObject3DMetaDataPresent* indicates whether the representation is using 3D objects. It is set to TRUE if the *m_ucObjRepresTypeIndex* is either:
  - REP_TYPE_3D_OBJECT_SINGLE_SRC_PER_WF; or
  - REP_TYPE_MONO_3D_OBJECT_MULTI_SRC_PER_WF.
- *m_bMonoObjWithMultipleSourcesFlag* indicates if 3D object representation corresponds to the use case with single waveform to which multiple 3D sources are assigned; it is set to TRUE if *m_ucObjRepresTypeIndex* is REP_TYPE_MONO_3D_OBJECT_MULTI_SRC_PER_WF.
- *m_bAmbisonicEncObjFlag* indicates if the representation is based on Ambisonic encoded signals; it is set to TRUE if *m_ucObjRepresTypeIndex* is REP_TYPE_AMBISONIC.

For object with *unObjectID* = 256, *m_bObject3DMetaDataPresent* and *the m_bMonoObjWithMultipleSourcesFlag* are both FALSE.

Table 7-23: Definition of Object Representation Types

Index	Gating Types	Mnemonic
0	Multi-channel representation in layout described by a channel mask	REP_TYPE_CH_MASK_BASED
1	Multi-channel representation in layout described by a channel mask obtained by rendering 2D (no height) content with spatial resolution than higher indicated by the encoded layout	REP_TYPE_MTRX2D_CH_MASK_BASED
2	Multi-channel representation in layout described by a channel mask obtained by rendering 3D (includes height) content with spatial resolution higher than indicated by the encoded layout	REP_TYPE_MTRX3D_CH_MASK_BASED
3	Binaurally processed audio (2 waveforms by default)	REP_TYPE_BINAURAL
4	Ambisonic representation	REP_TYPE_AMBISONIC
5	Audio tracks with associated mixing matrix to particular channel mask-based output layouts	REP_TYPE_AUDIO_TRACKS
6	3D Object with one 3D source associated per each waveform	REP_TYPE_3D_OBJECT_SINGLE_SRC_PER_WF
7	Mono 3D Object with multiple 3D sources associated to the same waveform	REP_TYPE_MONO_3D_OBJECT_MULTI_SRC_PER_WF

7.8.11.3 m_ucObjectImportanceLevel

This 3-bit field indicates the importance of an object on the scale from 0 (the lowest importance) to 7 (the highest importance).

In case a decoder implementation is presented with more objects than it can handle, the objects shall be first ordered from the highest to the lowest importance. Rendering shall start from objects with the highest importance and up to the limit on number of objects/waveforms imposed by the decoder implementation.

For full channel mask-based presentation (*unObjectID*==256) the *m_ucObjectImportanceLevel* = 7 by default.

7.8.11.4 bObjTypeDescrPresent

This 1-bit field indicates if the object type descriptor is transmitted in the stream (*bObjTypeDescrPresent* is TRUE) or an unknown object type is assumed by default (*bObjTypeDescrPresent* is FALSE).

7.8.11.5 m_unObjTypeDescrIndex

This variable length field is present in the stream only if the *bObjTypeDescrPresent* is TRUE. The *m_unObjTypeDescrIndex* is an index into a table of predefined object types according to the Definition of Object Types table (Table 7-10). The 4-bit or 6-bit codes are used to extract the *m_unObjTypeDescrIndex* in range from 0 to 7 or from 0 to 31 respectively.

For object with *unObjectID* = 256 the object type is not transmitted but is set, by default, to complete audio presentation type.

7.8.11.6 ucObjAudioChunkIndex

This variable length field indicates the index of an audio chunk that holds the audio waveforms associated with the object.

The variable length codes of length 2, 6, 7 or 11 bits are used to represent the audio chunk index values of up to 1, 17, 33 or 255 respectively.

For object with the *unObjectID* = 256, *m_ucObjAudioChunkIndex* is not transmitted but by default is set to 0.

7.8.11.7 m_ucObjNaviWitinACIndex

This variable length field indicates the object navigation within the audio chunk index. It is used by the audio decoders when given the audio chunk payload to navigate to the audio data corresponding to this object.

The variable length codes of length 4, 5, 7 or 11 bits are used to represent the navigation index values of up to 7, 15, 31 or 255 respectively.

When *unObjectID* = 256, *m_ucObjNaviWitinACIndex* is not transmitted and by default is set to 0.

7.8.11.8 m_bPerObjLTLoudnessMDPresent

If this 1-bit flag is TRUE, then per-object long term loudness metadata is present in the stream.

7.8.11.9 Per-Object Loudness Metadata

This block of metadata representing per-object long term loudness parameters is present in the stream only if *m_bPerObjLTLoudnessMDPresent* is TRUE. This metadata is not present for the object with *unObjectID*=256.

Long term loudness measurement data is transmitted only once in the interval between the two sync frames, i.e. in frames when *m_bObjStartFrameFlag* is TRUE. In all other frames the loudness parameters maintain the values extracted in the previous frame where *m_bObjStartFrameFlag* = TRUE.

This metadata is not present for the object with *unObjectID*=256.

Pseudo-code for extraction of this block of metadata is listed in Table 7-24.

Table 7-24: ExtractPerObjLTLoudnessMD

Syntax	Reference
`// Function to extract and inverse quantize object long term loudness parameters`	
`void CObjLTLoudnessMD:: ExtractObjectMetadata (void)`	
`{`	
`  // BS.1770 integrated loudness of each object`	
`  m_rObjIntegrLoudnessMsr =`	7.8.11.10.1
`  rPerObjLongTermLoudnessMeasure_Table[(uint) ExtractBits(6)];`	
`  // If this object is replacing some other object, TRUE indicates that this`	
`  // object should be normalized to the long-term level of the object it is`	
`  // replacing.`	
`  m_bMatchLDofReplacedObj = (ExtractBits(1)==1) ? TRUE : FALSE;`	7.8.11.10.2
`  // If TRUE try to normalize to the loudness level similar to the`	
`  // other objects of this type.`	
`  m_bMatchLDofSimilarObj = (ExtractBits(1)==1) ? TRUE : FALSE;`	7.8.11.10.3
`} // End of ExtractLTLoudnessMD() function`	

7.8.11.10 Parameters for ExtractObjectMetadata

7.8.11.10.1 m_rObjIntegrLoudnessMsr

This 6-bit field is a look-up index into the Per-Object Long Term Loudness Measure Table (clause 5.2.2.3). *m_rObjIntegrLoudnessMsr* represents the long term loudness measure for the object measured according to the Recommendation ITU-R BS.1770-4 [2].

7.8.11.10.2 m_bMatchLDofReplacedObj

This 1-bit flag, when TRUE, indicates that when this object is replacing some other object, this object shall be normalized to the long-term loudness of the object it is replacing.

7.8.11.10.3 m_bMatchLDofSimilarObj

This 1-bit flag, when TRUE, indicates that loudness management processing shall attempt to normalize this object to have the long term loudness level similar to the other objects of this type.

7.8.11.11 Object Interactivity Related Metadata

The content creator may mark certain object as interactive to allow the end user to modify certain object properties. The amount of modification that may be applied to certain object properties is described by parameters in this metadata block.

Pseudo-code that implements extraction of this metadata block is listed in Table 7-25.

Table 7-25: ExtractObjectInteractMD

Syntax	Reference
`void CObjInterLimits::ExtractObjectInteractMD(bool bInteractiveObjectsLimitsPresent,`	
`uint unObjectID, bool bObj3DMDPresent)`	
`{`	
`  uint unTemp;`	
`  if (unObjectID == 256)`	
`  {`	
`    // Make sure interactivity is not allowed`	
`    m_bObjInteractiveFlag = FALSE;`	
`    m_unMaxInterObjGainBoostdB = 0; // 0dB`	
`    m_unMaxInterObjGainAttendB = 0; // 0dB`	
`    m_unObjInterPosMaxDeltaAzim = 0; // 0 degrees`	
`    m_unObjInterPosMaxDeltaElev = 0; // 0 degrees`	
`  }`	
`  else`	
`  {`	
`    // If m_bObjInteractiveFlag is TRUE the object interaction is allowed`	
`    m_bObjInteractiveFlag = (ExtractBits(1) == 1) ? TRUE : FALSE;`	7.8.11.12.1
`    if (bInteractiveObjectsLimitsPresent) {`	
`      m_bObjInterLimitsFlag = (m_bObjInteractiveFlag) ? (bool)ExtractBits(1) : FALSE;`	7.8.11.12.2
`    }`	
`    else {`	
`      m_bObjInterLimitsFlag = FALSE;`	
`    }`	
`    if (m_bObjInterLimitsFlag) {`	
`      {`	
`        unTemp = ExtractBits(2);`	7.8.11.12.3
`        unTemp = (unTemp == 0) ? 0 : 3 * (1 << (unTemp - 1));`	
`        m_unMaxInterObjGainBoostdB = unTemp;`	
`      }`	
`      unTemp = ExtractBits(3);`	7.8.11.12.4
`      uint unTable[] = { 0, 3, 6, 9, 12, 20, 30, 200 };`	
`      m_unMaxInterObjGainAttendB = unTable[unTemp];`	
`      // Present only if bObj3DMDPresent=TRUE.`	
`      // One sided azimuth angle within which the object's centroid is allowed to`	
`      // move by interactive controls. Packed as a 3-bit index with corresponding`	
`      // values of 0, 15, 30, 45, 60, 90, 120 and 180 (full sphere). If not`	
`      // transmitted the default value is 0 implying that interactive position`	
`      // control is not allowed.`	
`      {`	
`      unTemp = (bObj3DMDPresent) ? (uint)ExtractBits(3) : (uint)0;`	7.8.11.12.5
`      uint unTable[] = { 0, 15, 30, 45, 60, 90, 120, 180 };`	
`      m_unObjInterPosMaxDeltaAzim = unTable[unTemp];`	
`      }`	
`      //  Present only if bObj3DMDPresent=TRUE.`	
`      //  One sided elevation angle within which the object's`	
`      //  centroid is allowed to move by interactive controls.`	
`      //  Packed as a 3-bit index with corresponding values of`	
`      //  0, 15, 30, 45, 60, 90, 120 and 180 (full sphere).`	
`      //  If not transmitted the default value is 0 implying that`	
`      //  interactive position control is not allowed.`	
`      //`	
`      {`	
`      unTemp = (bObj3DMDPresent) ? (uint)ExtractBits(3) : (uint)0;`	7.8.11.12.6
`      uint unTable[] = { 0, 15, 30, 45, 60, 90, 120, 180 };`	
`      m_unObjInterPosMaxDeltaElev = unTable[unTemp];`	
`      }`	
`    } // End of m_bObjInterLimitsFlag = TRUE case`	
`    else {`	
`      // If not transmitted the default value is 6dB for interactive objects.`	
`      if (m_bObjInteractiveFlag) {`	
`        m_unMaxInterObjGainBoostdB = 6; // 6dB`	
`      }`	
`      else {`	
`        m_unMaxInterObjGainBoostdB = 0; // 0 -> 0dB`	
`      }`	

Syntax	Reference

`// If not transmitted the default value is 0dB i.e. no attenuation allowed.`
`m_unMaxInterObjGainAttendB = 0; // 0dB`
`// If not transmitted the default value is 0 implying that interactive position control is not allowed.`
`m_unObjInterPosMaxDeltaAzim = 0; // 0 degrees`
`// If not transmitted the default value is 0 implying that interactive position control is not allowed.`
`m_unObjInterPosMaxDeltaElev = 0; // 0 degrees`
`} // End of m_bObjInterLimitsFlag = FALSE case`
`} // End of unObjectID != 256`
`} // End of ExtractObjectInteractMD() function`	

7.8.11.12 Parameters for ExtractObjectInteractMD

7.8.11.12.1 m_bObjInteractiveFlag

If this 1-bit field is TRUE, then object interaction is allowed i.e. the end user may be offered to modify certain object properties.

For channel-based full mix object (*unObjectID*=256) this flag is not transmitted in the stream and the *m_bObjInteractiveFlag* is by default set to FALSE.

7.8.11.12.2 m_bObjInterLimitsFlag

This 1-bit flag is transmitted in the stream only if *m_bObjInteractiveFlag* and *bInteractiveObjectsLimitsPresent* are both TRUE. Otherwise *m_bObjInterLimitsFlag* defaults to FALSE.

When the *m_bObjInterLimitsFlag* is TRUE, it indicates that the user modifications of certain object properties are constrained by the limit parameters transmitted in the stream. When the *m_bObjInterLimitsFlag* is FALSE, the limit parameters are set to their default values specified in the decoder.

7.8.11.12.3 m_unMaxInterObjGainBoostdB

This 2-bit field is present in the stream only if *m_bObjInterLimitsFlag* is TRUE. Extracted 2-bit index corresponds to the values *for m_unMaxInterObjGainBoostdB* of 0, 3, 6 and 12 dB. The value of m_*unMaxInterObjGainBoostdB* represents the maximum allowed interactive object level boost in dB.

If the *m_bObjInterLimitsFlag* is FALSE the *m_unMaxInterObjGainBoostdB* is set to:

- 6 dB if the *m_bObjInteractiveFlag* is TRUE; or
- 0 dB if the *m_bObjInteractiveFlag* is FALSE.

For channel-based full mix object (*unObjectID*=256) the *m_unMaxInterObjGainBoostdB* is not transmitted in the stream and is set by default to 0 dB.

7.8.11.12.4 m_unMaxInterObjGainAttendB

This 3-bit field is present in the stream only if the *m_bObjInterLimitsFlag* is TRUE. Extracted 3-bit index corresponds to the values for *m_unMaxInterObjGainAttendB* of 0, 3, 6, 9, 12, 20, 30 and ∞ (represented by 200) dB. The value of *m_unMaxInterObjGainAttendB* represents the maximum allowed interactive object level attenuation in dB.

If the *m_bObjInterLimitsFlag* is FALSE the *m_unMaxInterObjGainAttendB* is set to 0 dB.

For channel-based full mix object (*unObjectID*=256) the *m_unMaxInterObjGainAttendB* is not transmitted in the stream and is set by default to 0 dB.

7.8.11.12.5 m_unObjInterPosMaxDeltaAzim

This 3-bit field is present in the stream only if the *m_bObjInterLimitsFlag* is TRUE. Extracted 3-bit index corresponds to the values for *m_unObjInterPosMaxDeltaAzim* of 0, 15, 30, 45, 60, 90, 120 and 180 degrees, in ascending order.

The value of *m_unObjInterPosMaxDeltaAzim* represents an azimuth angle in degrees. User controls are allowed to modify the azimuth angle of object's centroid by ± *m_unObjInterPosMaxDeltaAzim* degrees around its nominal position.

If the *m_bObjInterLimitsFlag* is FALSE then *m_unObjInterPosMaxDeltaAzim* is set to 0 degrees effectively preventing modification of the nominal azimuth angle of an object centroid.

For channel-based full mix object (*unObjectID*=256), *m_unObjInterPosMaxDeltaAzim* is not transmitted in the stream and by default set to 0 degrees.

7.8.11.12.6 m_unObjInterPosMaxDeltaElev

This 3-bit field is present in the stream only if *m_bObjInterLimitsFlag* is TRUE. Extracted 3-bit index corresponds to the values for *m_unObjInterPosMaxDeltaElev* of 0, 15, 30, 45, 60, 90, 120 and 180 degrees, in ascending order.

The value of *m_unObjInterPosMaxDeltaElev* represents an elevation angle in degrees. User controls are allowed to modify the elevation angle of object's centroid by ± *m_unObjInterPosMaxDeltaElev* degrees around its nominal position.

If the *m_bObjInterLimitsFlag* is FALSE the *m_unObjInterPosMaxDeltaElev* is set to 0 degrees effectively preventing modification of the nominal elevation angle of an object centroid.

For channel-based full mix object (*unObjectID*=256), *m_unObjInterPosMaxDeltaElev* is not transmitted in the stream and by default set to 0 degrees.

7.8.11.13 Metadata for Channel Mask Parameters

This metadata block is present in the stream only in frames where the *bObjStartFrameFlag* is TRUE and only for objects where the m_*bChMaskObjectFlag* is TRUE.

Pseudo-code for extraction of this block of metadata is listed in Table 7-26.

Table 7-26: ExtractChMaskParams

Syntax	Reference
`void CL3DMD::ExtractChMaskParams(void)`	
`{`	
`  m_ucChLayoutIndex = (m_ucObjRepresTypeIndex == REP_TYPE_BINAURAL) ? 1 :`	
`(uchar)ExtractBits(4);`	7.8.11.14.1
`  //`	
`  switch (m_ucChLayoutIndex)`	
`  {`	
`  case 14:`	
`    m_uint32ChActivityMask = (uint)ExtractBits(16);`	7.8.11.14.2
`    break;`	
`  case 15:`	
`    m_uint32ChActivityMask = (uint)ExtractBits(32);`	
`    break;`	
`  default:`	
`    m_uint32ChActivityMask = unChMask_Index_Table[m_ucChLayoutIndex];`	Table 7-27
`    break;`	
`  } // End of unChLayoutIndex switch`	
`  m_ucNumWaveFormsInObj = (uchar)GetNumSpeakersInLayout(m_uint32ChActivityMask);`	7.8.11.14.3
`  m_bObjectRendExceptPresent = (m_ucChLayoutIndex < 2) ? FALSE : (bool)ExtractBits(1);`	7.8.11.20
`}  // End of ExtractChMaskParams() function`	

7.8.11.14 Parameters for ExtractChMaskParams

7.8.11.14.1 m_ucChLayoutIndex

This 4-bit field represents an index into a look-up table with channel activity mask based layout descriptions as shown in Table 7-27.

Table 7-27: unChMask_Index_Table

ucChLayoutIndex	Channel Layout Description	32-bit Channel Activity Mask
0	C (1.0)	0x00000001
1	LR (2.0)	0x00000002
2	LR + LsRs (4.0)	0x00000006
3	C+LR+LsRs+LFE1 (5.1)	0x0000000F
4	C + LR + LsRs + LFE1 + Cs (6.1)	0x0000001F
5	C+LR+LssRss+LsrRsr+LFE1 (7.1)	0x0000084B
6	C+LR+LsRs+LhRh+LFE1 (5.1+LhRh)	0x0000002F
7	C+LR+LsRs+LhRh+LhrRhr+LFE1 (9.1)	0x0000802F
8	C+LR+LssRss+LsrRsr+LhRh+Chr+LFE 1 (10.1)	0x0000486B
9	C+LR+LssRss+LsrRsr+LhRh+LhrRhr+LFE1 (11.1)	0x0000886B
10	NHK 22.2 Layout (22.2)	0x0003FBFB
11	CLR (3.0)	0x00000003
12	C+LR+LsRs(5.0)	0x00000007
13	C+LR+LssRss(LsRs)+LsrRsr (7.0)	0x00000843
14	16-bit Channel Activity Mask Explicitly Transmitted	16-bit Mask
15	32-bit Channel Activity Mask Explicitly Transmitted	32-bit Mask

7.8.11.14.2 m_uint32ChActivityMask

This field is transmitted in the stream only if *m_bChMaskObjectFlag* is TRUE. For values of *m_ucChLayoutIndex* between 0 and 13 the associated channel activity mask *m_uint32ChActivityMask* is extracted directly from Table 7-28.

For other values of *m_ucChLayoutIndex* the channel activity mask *m_uint32ChActivityMask* is extracted from the stream as:

- 16-bit unsigned integer mask when the *m_ucChLayoutIndex*=14; or
- 32-bit unsigned integer mask when the *m_ucChLayoutIndex*=15.

The associated channel layout is defined by the *m_uint32ChActivityMask* according to Table 7-28.

Table 7-28: Mapping Channels to Bits within Channel Activity Mask

Speaker Name	Description	Mask bit	Number of channels	Nominal Position (see note)
C	Center in front of listener	0x00000001	1	0, 0
LR	Left/Right in front	0x00000002	2	±30, 0
LsRs	Left/Right surround on side in rear	0x00000004	2	±110, 0
LFE1	Low frequency effects subwoofer	0x00000008	1	NA
Cs	Center surround in rear	0x00000010	1	180, 0
LhRh	Left/Right height in front	0x00000020	2	±45, 45
LsrRsr	Left/Right surround in rear	0x00000040	2	±150, 0
Ch	Center Height in front	0x00000080	1	0, 45
Oh	Over the listener's head	0x00000100	1	0, 90
LcRc	Between left/right and center in front	0x00000200	2	±15, 0
LwRw	Left/Right on side in front	0x00000400	2	±60, 0
LssRss	Left/Right surround on side	0x00000800	2	±90, 0
LFE2	Second low frequency effects subwoofer	0x00001000	1	NA
LhsRhs	Left/Right height on side (or Left/Right Top in middle)	0x00002000	2	±90, 45
Chr	Center height in rear	0x00004000	1	180, 45
LhrRhr	Left/Right height in rear	0x00008000	2	±135, 45
Cbf	Center below in front	0x00010000	1	0, -30
LbfRbf	Left/Right below in front	0x00020000	2	±45, -30
LtfRtf	Left/Right top in front	0x00040000	2	±45, 60
LtrRtr	Left/Right top in rear	0x00080000	2	±135, 60
	Reserved	0x00100000 - 0x80000000		
NOTE: Nominal position provided in degrees (azimuth, elevation) relative to the unit sphere around a listener positioned at the centre.				

7.8.11.14.3 Count the Number of Active Channels

The helper function *GetNumSpeakersInLayout()* calculates the number of active channels in a channel mask-based on the number of channels represented per flag as defined in Table 7-29.

Table 7-29: GetNumSpeakersInLayout

Syntax

```
const uchar ucNumChPerChMaskBit_Table[] = {
1, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2, 2, 1, 2, 1, 2, 1, 2, 2, 2
};

uchar GetNumSpeakersInLayout(uint unChLayoutMask)
{
  uchar ucNumCh = 0;
  for (uint nb = 0; nb < 32; nb++) {
    // Add up the number of channels corresponding to the Channel Mask
    if (((unChLayoutMask >> nb) & 0x1) == 1) {
      ucNumCh += ucNumChPerChMaskBit_Table[nb];
    }
  }
  return ucNumCh;
}
```

7.8.11.15 Per Object Renderer Configuration Parameters

This field is transmitted in the stream only if *m_bObject3DMetaDataPresent* = TRUE. Pseudo-code for extraction of per-object 3D renderer configuration parameters is shown in Table 7-30.

Table 7-30: ExtractRendererConfigParams

Syntax	Reference
`void CL3DMD::ExtractRendererConfigParams(void)`	
`{`	
`  m_bObjectRendExceptPresent = (ExtractBits(1) == 1) ? TRUE : FALSE;`	7.8.11.20
`  m_bEnhancedREConfigParamsPresent = (ExtractBits(1) == 1) ? TRUE : FALSE;`	7.8.11.16.1
`  if (m_bEnhancedREConfigParamsPresent)`	
`  {`	
`    m_bUseDivergApperObjSpread = (ExtractBits(1) == 1) ? TRUE : FALSE;`	7.8.11.16.2
`    m_bEnableSnaptoSpkrs = (ExtractBits(1) == 1) ? TRUE : FALSE;`	7.8.11.16.3
`    uint unTemp = (m_bEnableSnaptoSpkrs) ? 12 * ExtractBits(5) : 0;`	
`    m_unSnap2SpkrSpherCapAngle = (unTemp > 360) ? 360 : unTemp;`	7.8.11.16.4
`    uchar uctable[4] = { 0, 12, 16, 20 };`	
`    m_unRendrerExcludedSpkrsChMask = (uint)ExtractVarLenBitFields(uctable, false);`	7.8.11.16.5
`    m_bAudioObjLinked2VisualObj = (ExtractBits(1) == 1) ? TRUE : FALSE;`	7.8.11.16.6
`  } // End of case when the m_bEnhancedREConfigParamsPresent is TRUE`	
`  else`	
`  {`	
`    // Initialize parameters that were not transmitted`	
`    m_bUseDivergApperObjSpread = FALSE;`	
`    m_bEnableSnaptoSpkrs = FALSE;`	
`    m_unSnap2SpkrSpherCapAngle = FALSE;`	
`    m_unRendrerExcludedSpkrsChMask = 0;`	
`    m_bAudioObjLinked2VisualObj = FALSE;`	
`  }`	
`} // End of ExtractRendererConfigParams()`	

7.8.11.16 Parameters for ExtractRendererConfigParams

7.8.11.16.1 m_bEnhancedREConfigParamsPresent

This 1-bit flag, when TRUE, indicates that an enhanced set of metadata parameters for the configuration of the rendering engine is present in the stream. Some of these parameters may not be used by the basic rendering engine implementation (OBJECT_RENDERER_TYPE_BASIC) but may be used by the extended renderer implementation (OBJECT_RENDERER_TYPE_EXTENDED).

7.8.11.16.2 m_bUseDivergApperObjSpread

This 1-bit flag is present in the stream if *m_bEnhancedREConfigParamsPresent* = TRUE and *m_bObject3DMetaDataPresent* = TRUE. Its value distinguishes two different interpretations of the object extent related parameters *m_3DSrcSpreadParamOne* and *m_3DSrcSpreadParamTwo*.

7.8.11.16.3 m_bEnableSnaptoSpkrs

This 1-bit flag indicates if snap to the nearest speaker shall be enabled for this object (when *m_bEnableSnaptoSpkrs* is TRUE).

7.8.11.16.4 m_unSnap2SpkrSpherCapAngle

This 5-bit field is present in the stream only if *m_bEnableSnaptoSpkrs* is TRUE. It represents an angle in degrees, which defines a spherical cap around the object's centroid. If *m_unSnap2SpkrSpherCapAngle* is greater than 0, the object will be snapped to the speaker within the cap nearest to the object's centroid. If no speakers are enclosed within the cap, the object will be rendered without snapping.

Valid values for the *m_unSnap2SpkrSpherCapAngle* range from 0 to 360 degrees in steps of 12 degrees:

m_unSnap2SpkrSpherCapAngle=0 implies no snapping

m_unSnap2SpkrSpherCapAngle=360 guarantees snapping to the nearest speaker

7.8.11.16.5 m_unRendrerExcludedSpkrsChMask

This variable length field represents a channel mask that defines which speakers shall not receive any contribution from this object. Each bit within the *m_unRendrerExcludedSpkrsChMask* corresponds to specific speaker(s) according to Table 7-28. Code lengths of 1, 14, 19 and 23 bits are used to extract masks of 0, 12, 16 or 20 bits. Note that when *m_unRendrerExcludedSpkrsChMask*=0, it implies that no speakers are excluded from receiving a contribution from this object.

7.8.11.16.6 m_bAudioObjLinked2VisualObj

This 1-bit flag when TRUE, indicates that this audio object is linked to a visual object. If *m_bAudioObjLinked2VisualObj* is TRUE and the object position is within the reference screen, the rendering engine shall attempt to restrict the rendering of this object such that it subjectively stays in the area of the playback screen.

7.8.11.17 m_ucNumWaveFormsInObj

This field is transmitted in the stream only if *m_bObject3DMetaDataPresent* = FALSE. This variable length field indicates the number of waveforms that are defined for this object. Values up to 2, 10, 18 or 32 are transmitted using 2, 5, 6 or 7 bit variable length code.

7.8.11.18 Metadata Describing Ambisonic Representation

This field is transmitted in the stream only if *m_bObject3DMetaDataPresent* = FALSE. This block of metadata is transmitted in the stream if *m_bAmbisonicEncObjFlag* is TRUE.

This block of metadata contains parameters that describe an Ambisonic encoded representation. By default, the ordering and normalization of the Ambisonic signals is according to the AmbiX convention. In particular the spherical harmonics are packed according to their order (*n*) and degree (*m*) using the Ambisonic Channel Number (ACN) index:

$$ACN = n^2 + n + m \qquad \text{where} \begin{cases} 0 \le n \le N \\ -n \le m \le n \end{cases}$$

and where N is the final order of the Ambisonic representation.

By default, the representation order can be calculated directly from the number of transmitted signals *ucNumWaveFormsInObj* as:

$$N = \left\lfloor \sqrt{ucNumWaveFormsInObj} \right\rfloor$$

The assumed normalization of the signals is SN3D [3].

Ambisonic signals that are encoded with different ordering, partial representations (missing harmonics) and/or a different normalization can be accommodated by use of an adaptor matrix AM that may be either be predefined or transmitted in the bitstream.

In particular a column vector of T=*ucNumWaveFormsInObj* transmitted signals at the time instance *k* is denoted by:

$$X(k) = [x1(k) \cdot x2(k) \cdot \cdots \cdot XT(k)]'$$

column vector of $LL = (N+1)^2$ entries, representing samples of Ambisonic signals up to the N-th order according to the AmbiX convention at time instance *k*, by:

$$Y(k) = [y1(k) \cdot y2(k) \cdot \cdots \cdot yLL(k)]'$$

The desired signal vector *Y(k)* can be obtained from *X(k)* and adaptor matrix *AM* as:

$$Y(k) = AM * X(k)$$

Pseudo-code for extraction of this block of metadata is listed in Table 7-31.

Table 7-31: m_nBFormatAmbiAdMtrx

Syntax	Reference

// Predefined Static Ambisonic Adapter Matrices
 	
// Adapter Matrix Representing W, X, Y and Z signals of Ambisonic B Format
int CAmbisonicReprMD::m_nBFormatAmbiAdMtrx[16] =
{
SQRT2Q1DOT15, 0, 0, 0,
0, 0, ONE_IN_Q15, 0,
0, 0, 0, ONE_IN_Q15,
0, ONE_IN_Q15, 0, 0
};
 	
// Adapter Matrix Representing W, X and Y of B Format
int CAmbisonicReprMD::m_nHorizBAmbiAdMtrx[12] =
{
SQRT2Q1DOT15, 0, 0,
0, 0, ONE_IN_Q15,
0, 0, 0,
0, ONE_IN_Q15, 0
};
 	
// Function for Extraction of Ambisonic Adapter Matrix
void CAmbisonicReprMD::ExtractAmbisonicsMD(uchar ucNumWaveFormsInObj)
{
int nTemp;
int nSign;
uint nuWfToACNContribMask;
 	
// Indicates presence of an Adaptor matrix or use one of the default Adaptor matrices
 	
m_ucAmbixAdaptorMtrxIndicator = (uchar) ExtractBits(3);
	7.8.11.19.1

switch (m_ucAmbixAdaptorMtrxIndicator) {
case 0:
// No Adaptor matrix is needed.
// Signals are normalized and ordered according to AmbiX specification
m_ucAmbisonicRepresOrder = (uchar(sqrt((float)ucNumWaveFormsInObj))) - 1;
if ((m_ucAmbisonicRepresOrder + 1)*(m_ucAmbisonicRepresOrder + 1) !=
ucNumWaveFormsInObj)
exit(-1); // Exit with an error
 	
// This is full mth order representation that can be denoted as "mHmV"
m_ucAmbisonicRepresOrderVorP = m_ucAmbisonicRepresOrder;
break;
case 1:
// Horizontal only B format i.e. W, X, and Y
if (ucNumWaveFormsInObj != 3) {
exit(-1); // Exit with an error
}
m_ucAmbisonicRepresOrder=1;
m_ucAmbisonicRepresOrderVorP = 0;
m_unNumACNSignals = 4;
// Allocate memory for adapter matrix
if (m_pAmbiAdaptorMtrx==NULL)
{
m_pAmbiAdaptorMtrx = new int [m_unNumACNSignals * ucNumWaveFormsInObj];
memcpy(m_pAmbiAdaptorMtrx,m_nHorizBAmbiAdMtrx, m_unNumACNSignals *
ucNumWaveFormsInObj * sizeof(int));
}
break;
 	
case 2:
// B format i.e. W, X, Y and Z
if (ucNumWaveFormsInObj != 4)
{
exit(-1); // Exit with an error
}
m_ucAmbisonicRepresOrder = 1;
m_ucAmbisonicRepresOrderVorP = 1;
m_unNumACNSignals = 4;
if (m_pAmbiAdaptorMtrx==NULL)
{
m_pAmbiAdaptorMtrx = new int[m_unNumACNSignals * (uint) ucNumWaveFormsInObj];
memcpy(m_pAmbiAdaptorMtrx,m_nHorizBAmbiAdMtrx, m_unNumACNSignals *
(uint) ucNumWaveFormsInObj * sizeof(int));
}
Break;	

Syntax	Reference

case 3:
// Adaptor matrix is transmitted in the stream. The size of the matrix is
// (N+1)^2 x ucNumWaveFormsInObj where N is the order of Ambisonic representation.
// In this case N can't be determined from ucNumWaveFormsInObj so it is sent in the
// bitstream.
 	
m_ucAmbisonicRepresOrder = (uchar) ExtractBits(2)+1;
	7.8.11.19.2

// Calculate number of rows m_unNumACNSignals as (N+1)^2
m_unNumACNSignals = (uint) m_ucAmbisonicRepresOrder + 1;
m_unNumACNSignals = m_unNumACNSignals*m_unNumACNSignals;
 	
// Allocate memory for Adaptor matrix
if (m_pAmbiAdaptorMtrx==NULL)
{
m_pAmbiAdaptorMtrx = new int[m_unNumACNSignals * (uint) ucNumWaveFormsInObj];
}
/* Loop over columns of Adaptor matrix */
for (uchar nco=0, uncount=0; nco<ucNumWaveFormsInObj; nco++)
{
// Since Adaptor matrix is generally very sparse for each column of the matrix
// we transmit a bit mask nuWfToACNContribMask that points to the non-zero
// elements. Only the non-zero elements are explicitly transmitted in the stream.
nuWfToACNContribMask = (uint) ExtractBits(m_unNumACNSignals);
	7.8.11.19.3

for (uint nro=0; nro<m_unNumACNSignals; nro++) // Row loop
{
// First row corresponds to the LSB in the mask
if ((nuWfToACNContribMask>>nro) & 0x1) {
// extract the sign and magnitude of the scale factor
nSign = (int) ((ExtractBits(1)==1) ? -1 : 1);
nTemp = (int) unScaleFactorsTable[(uchar) ExtractBits(6)]; //
// Prior to transmission entries were scaled by 2 to fit into Q0.15
// format. Translate back from Q0.15 to Q1.15 and apply the sign.
nTemp = nSign * (nTemp<<1);
}
else{
nTemp = 0; // 0 -> no contribution (null scale factor)
}
// Entries of the matrix are signed integers in Q1.15 format
m_pAmbiAdaptorMtrx[uncount++] = nTemp;
} // End of row loop
} // End of column loop
break; // End of case 3
	7.8.11.19.4

case 4:
// In this case transmitted signals are in m-th order horizontal only AmbiX format
// denoted as "mH0V". The number of transmitted signals is equal to 2*m+1.
// The m is calculated as m = (ucNumWaveFormsInObj-1)/2.
m_ucAmbisonicRepresOrder = (ucNumWaveFormsInObj - 1) >> 2;
if ((2 * m_ucAmbisonicRepresOrder + 1) != ucNumWaveFormsInObj)
{ exit(-1); // Exit with error
}
m_ucAmbisonicRepresOrderVorP = 0;
 	
// Calculate number of rows of Adopter matrix m_unNumACNSignals as
// (m_ucAmbisonicRepresOrder+1)^2
m_unNumACNSignals = m_ucAmbisonicRepresOrder + 1;
m_unNumACNSignals = m_unNumACNSignals*m_unNumACNSignals;
 	
// Determine ACN of each transmitted harmonic and construct Ambisonic Adopter matrix
// for underlying mH0V mixed order representation
ConstructAdopterMtrxformHnVorPRepres (ucNumWaveFormsInObj, true);
break; // End of case 4	

case 5:
// Transmitted signals are in "mH1P" mixed order AmbiX format.
// The number of transmitted signals is equal to 2*m+2 where there are
// 2*m+1 horizontal only components and one extra signal corresponds to the
// "Z" component (ACN=2).
 	
// The m is calculated as m = (ucNumWaveFormsInObj-2)/2.
m_ucAmbisonicRepresOrder = (ucNumWaveFormsInObj - 2) >> 1;
if ((2 * m_ucAmbisonicRepresOrder + 2) != ucNumWaveFormsInObj) {
exit(-1); // Exit with an errror
}
m_ucAmbisonicRepresOrderVorP = 0;
// Calculate number of rows of Adopter matrix m_unNumACNSignals
// as (m_ucAmbisonicRepresOrder+1)**2
m_unNumACNSignals = (uint) m_ucAmbisonicRepresOrder + 1;
m_unNumACNSignals = m_unNumACNSignals*m_unNumACNSignals;	

Syntax	Reference

```
    // Determine ACN of each transmitted harmonic and construct Ambisonic Adopter matrix
    // for underlying mH1P mixed order representation
    ConstructAdopterMtrxformHnVorPRepres(ucNumWaveFormsInObj, false);
    break; // End of case 5
  case 6:
    //   Transmitted signals are in "mHnV" mixed order AmbiX format. The number of
    //  transmitted signals is equal to (m+1)^2 - (m-n)^2 where m > n.
    // m and n are obtained by table look-up using the ucNumWaveFormsInObj
    // as an input parameter.

    ExtractOrderParamsformHnVRepres(ucNumWaveFormsInObj);
    // Determine ACN of each transmitted harmonic and construct Ambisonic Adopter matrix
    // for underlying mHnV mixed order representation
    ConstructAdopterMtrxformHnVorPRepres(ucNumWaveFormsInObj, true);
    break; // End of case 6
  case 7:
    exit(-1); Exit with an error
    break; // End of case 7
  } // End of switch
} // End of ExtractAmbisonicsMD()

// Helper functions for ExtractAmbisonicsMD ()

// Helper function to determine ACN of each transmitted harmonic and construct Ambisonic
// Adopter matrix for underlying "mHnV" or "mHnP" mixed order representation

void CAmbisonicReprMD::ConstructAdopterMtrxformHnVorPRepres(uchar ucNumWaveFormsInObj,
bool bmHnVFlag)
{
  // m_ucAmbisonicRepresOrder, representing number of rows in the Adoper matrix,
  // m, represented by m_ucAmbisonicRepresOrder and
  // n, represented by m_ucAmbisonicRepresOrderVorP
  // are all set during the extraction of m_ucAmbixAdaptorMtrxIndicator.

  uint unNumActiveHarmonics;
  if (bmHnVFlag)    // mHnV case
  {
    // Calculate number of active harmonics as (m+1)**2 - (m-n)**2
    unNumActiveHarmonics = (uint)(pow(float(m_ucAmbisonicRepresOrder + 1), 2.0)
      - pow(float(m_ucAmbisonicRepresOrder - m_ucAmbisonicRepresOrderVorP), 2.0));
  }
  else   // mHnP case
  {
    // Calculate number of active harmonics as 2*m+1 + n**2
    unNumActiveHarmonics = (uint)((float)(2*m_ucAmbisonicRepresOrder+1) +
    pow(float(m_ucAmbisonicRepresOrderVorP), 2.0));
  }
  if (unNumActiveHarmonics != ucNumWaveFormsInObj)
    exit(-1); // Exit with an error

  // Allocate memory for Adaptor matrix
  if (m_pAmbiAdaptorMtrx == NULL)
  m_pAmbiAdaptorMtrx = new int[m_unNumACNSignals * (uint)ucNumWaveFormsInObj];

  // Allocate memory for array holding ACN of each transmitted/active harmonic
  if (m_punActiveHarmonicsACNs == NULL)
  m_punActiveHarmonicsACNs = new uint[unNumActiveHarmonics];

  uint *punActiveHarmonics = m_punActiveHarmonicsACNs;
  int l2pl;

  if (bmHnVFlag)    // mHnV case
  {
    // Set ACN for each Active harmonic component of unmderlying mHnV representation

    int nbound;
    *punActiveHarmonics++ = 0; // Harmonic "W" with ACN = 0 is always included
    for (int l = 0; l < (int)m_ucAmbisonicRepresOrder; l++)
    {
      // For the current order l the active harmonics are -l, ... min(-l+m,0)
      // and max(1,l-m), ... l
      l2pl = l*(l + 1);
      nbound = (int)fmin((float)(-l + m_ucAmbisonicRepresOrderVorP), 0.0f);
      for (int k = -l; k < nbound; k++){
        *punActiveHarmonics++ = l2pl + k; // ACN = l**2 + l + k
      }
```

Syntax	Reference

```
      nbound = (int)fmax(1.0f, (float)(l - m_ucAmbisonicRepresOrderVorP));
      for (int k = nbound; k < l; k++){
        *punActiveHarmonics++ = l2pl + k; // ACN = l**2 + l + k
      }
    }
  }
  else    // mHnP case
  {
    // Set ACN for each Active harmonic component in unmderlying mHnP representation
    int nstride;
    *punActiveHarmonics++ = 0; // Harmonic "W" with ACN = 0 is always included
    for (int l = 0; l < (int)m_ucAmbisonicRepresOrder; l++){
      // If current order l is <= m_ucAmbisonicRepresOrderVorP all harmonics
      // from -l to l are selected
      // otherwise only -l and +l harmonics are selected
      nstride = (l>m_ucAmbisonicRepresOrderVorP) ? 2 * l : 1;
      l2pl = l*(l + 1);
      for (int k = -l; k < l; k = k + nstride){
        *punActiveHarmonics++ = l2pl + k; // ACN = l**2 + l + k
      }
    }
  }
  // Construct Adopter matrix that has unNumActiveHarmonics colums each with
  // m_unNumACNSignals elements. In each column k, set all except the one element to 0
  // where the only non-zero element is equal to ONE_IN_Q15 and it is in the
  // m_punActiveHarmonicsACNs[k] position within the column k.
  int nEntry;
  for (uint ncol = 0; ncol < unNumActiveHarmonics; ncol++){
    for (uint nrow = 0; nrow < m_unNumACNSignals; nrow++){
      nEntry = nrow*unNumActiveHarmonics + ncol;
      m_pAmbiAdaptorMtrx[nEntry] =
                                (nrow == m_punActiveHarmonicsACNs[ncol]) ? ONE_IN_Q15 : 0;
    }
  }
} // end of ConstructAdopterMtrxformHnVorPRepres
```

Syntax	Reference

```
// Helper function to extract orders m and n of Ambisonic representation of form "mHnV"

// For the case of "mHnV" representation: the 3x9 look-up matrix for extracting the
// Ambisonic representation order parameters m and n based on ucNumWaveFormsInObj

uchar CAmbisonicReprMD::m_ucNumCompOrderHOrderVMtrx[27] =
{
  8, 12, 15, 16, 20, 21, 24, 27, 32, // First row corresponds to the ucNumWaveFormsInObj
  2, 3, 3, 4, 5, 4, 4, 5, 5, // Second row corresponds to m
  1, 1, 2, 1, 1, 2, 3, 2, 3 // Third row corresponds to n
};
void CAmbisonicReprMD::ExtractOrderParamsformHnVRepres(uchar ucNumHarmComp)
{
  // There are 9 supported mixed order "mHnV" representations that are tabulated in
  // m_ucNumCompOrderHOrderVMtrx. Elements 0:8 represent valid number of harmonic
  // components. Find representation index that has the number of harmonic components
  // equal to ucNumHarmComp

  uchar ucNumRepres = sizeof(CAmbisonicReprMD::m_ucNumCompOrderHOrderVMtrx) /
    (3 * (sizeof(m_ucNumCompOrderHOrderVMtrx[1])));
  uchar ucRepresIndex = ucNumRepres;
  for (uchar ir = 0; ir++, ir < ucNumRepres;){
    if (m_ucNumCompOrderHOrderVMtrx[ir] == ucNumHarmComp)
    {
      ucRepresIndex = ir;
      break;
    }
  }
  if (ucRepresIndex == ucNumRepres){
    exit(-1); //error condition
  }
  // Extract parameter m of mHnV representation as ucRepresIndex element
  // of the second row of m_ucNumCompOrderHOrderVMtrx
  m_ucAmbisonicRepresOrder = m_ucNumCompOrderHOrderVMtrx[ucNumRepres + ucRepresIndex];

  // Extract parameter n of mHnV representation as ucRepresIndex element
  // of the third row of m_ucNumCompOrderHOrderVMtrx
  m_ucAmbisonicRepresOrderVorP =
                        m_ucNumCompOrderHOrderVMtrx[2*ucNumRepres + ucRepresIndex];

  // Calculate number of rows of Adopter matrix m_unNumACNSignals as
  // (m_ucAmbisonicRepresOrder+1)**2
  m_unNumACNSignals = (uint) m_ucAmbisonicRepresOrder + 1;
  m_unNumACNSignals = m_unNumACNSignals*m_unNumACNSignals;}
} // end of ExtractOrderParamsformHnVRepres()
```

7.8.11.19 Parameters for ExtractAmbisonicsMD

7.8.11.19.1 m_ucAmbixAdaptorMtrxIndicator

This 3-bit filed indicates whether an adapter matrix shall be used for translation of transmitted signals into the signals in AmbiX format. In particular when:

- *m_ucAmbixAdaptorMtrxIndicator*=0: No adaptor matrix is needed; Ambisonic signals are transmitted according to the default ordering and normalization as per AmbiX specification [3].
- *m_ucAmbixAdaptorMtrxIndicator*=1: Transmitted signals are in horizontal only B format (W, X and Y); to translate the transmitted signals into the AmbiX compliant signals the *m_nHorizBAmbiAdMtrx[]* adaptor matrix shall be used.
- *m_ucAmbixAdaptorMtrxIndicator*=2: Transmitted signals are in B format (W, X, Y and Z); to translate the transmitted signals into the AmbiX compliant signals the *m_nBFormatAmbiAdMtrx[]* adaptor matrix shall be used.
- *m_ucAmbixAdaptorMtrxIndicator*=3: Custom adaptor matrix is transmitted in the bitstream to allow mapping of the transmitted Ambisonic signals into the AmbiX compliant Ambisonic signals.

- *m_ucAmbixAdaptorMtrxIndicator* = 4: Transmitted signals are in *m-th* order horizontal only AmbiX format denoted as "mH0V". The number of transmitted signals is equal to 2**m*+1. The *m* is calculated as *m* = (*ucNumWaveFormsInObj*-1)/2. To translate the transmitted signals into the AmbiX compliant signals an algorithmically calculated Adaptor matrix shall be used.

- *m_ucAmbixAdaptorMtrxIndicator* = 5: Transmitted signals are in "mH1P" mixed order AmbiX format. The number of transmitted signals is equal to 2**m*+2 where there are 2**m*+1 horizontal only components and one extra signal corresponds to the "*Z*" component (ACN=2).
  The *m* is calculated as *m* = (*ucNumWaveFormsInObj*-2)/2. To translate the transmitted signals into the AmbiX compliant signals an algorithmically calculated Adaptor matrix shall be used.

- *m_ucAmbixAdaptorMtrxIndicator* = 6: Transmitted signals are in "mHnV" mixed order AmbiX format. The number of transmitted signals is equal to $(m+1)^2 - (m-n)^2$ where $m > n$. The *m* and *n* are obtained by table look-up using *ucNumWaveFormsInObj* as an input parameter. The pseudo-code that implements this table look-up is listed in Table 7-31. To translate the transmitted signals into the AmbiX compliant signals an algorithmically calculated Adaptor matrix shall be used.

- *m_ucAmbixAdaptorMtrxIndicator* = 7: Reserved fo future use.

Note that the adapter matrix is used only when the Ambisonic signals are to be exported to the external Ambisonic decoder for further processing. If the Ambisonic signals are rendered into an output multi-channel layout using the associated rendering exception matrix (REM), the adapter matrix is ignored. This is because the elements of REM already include the effect of both the adapter matrix and the Ambisonic decoder matrix for particular output layout.

7.8.11.19.2 m_ucAmbisonicRepresOrder

This 2-bit field is transmitted only if an adaptor matrix is transmitted in the stream (*m_ucAmbixAdaptorMtrxIndicator*=3) and it represents the order of Ambisonic representation.

The size of transmitted adaptor matrix is $(N + 1)^2$ * *ucNumWaveFormsInObj* where the *N* is the order of Ambisonic representation. In this case *N* cannot be concluded from *ucNumWaveFormsInObj* but will be transmitted as *N* = *m_ucAmbisonicRepresOrder*. Allowed values for *m_ucAmbisonicRepresOrder* are 1, 2, 3 and 4.

7.8.11.19.3 nuWfToACNContribMask

This *m_unNumACNSignals*-bit mask is transmitted in the stream only if the *m_ucAmbixAdaptorMtrxIndicator*=3 where:

$$m_unNumACNSignals = (m_ucAmbisonicRepresOrder + 1)^2.$$

Since adaptor matrix is generally very sparse for each column of the matrix the *nuWfToACNContribMask* is transmitted to point to the non-zero elements. Only the non-zero elements of adaptor matrix are explicitly transmitted in the stream.

7.8.11.19.4 Non-zero Entries of the Adaptor Matrix

A non-zero entry in adaptor matrix is identified by the corresponding bit within the *nuWfToACNContribMask* being set to "1". Each non-zero entry is extracted in two parts:

- 1-bit field indicating the sign where:
  - 1 implies negative sign (sign_value = -1); and
  - 0 implies positive sign (sign_value = +1).
- 6-bit field representing an index (*unIndex*) for look-up into a table with scale factors represented in unsigned Q0.15 fixed point format as shown in Scale Factors Table (clause 5.2.2.1).

Entries of the adaptor matrix (a_{ij}) are signed integers in Q1.15 fixed point format. The values extracted from the scale table are translated to Q1.15 fixed point format as a_{ij} = *sign_value* * 2 * *unScaleFactorsTable* [unIndex].

7.8.11.20 m_bObjectRendExceptPresent

m_bObjectRendExceptPresent is interpreted according to the specific object type in context, but in all cases, when true, indicates the presence of additional rendering exception metadata.

- In the case of an Ambisonic object, if *m_bAmbisonicEncObjFlag* is TRUE, then *m_bObjectRendExceptPresent* is present in the stream. If *m_bObjectRendExceptPresent* is TRUE, then the Ambisonic decoding matrices for up to four explicit output layouts may be transmitted in the stream.

  In the nominal case, an object encoded into Ambisonic signals is decoded by a dedicated Ambisonic decoder that is capable of decoding the Ambisonic signals without transmitted decoding matrices. However, it is possible to also provide the decoding matrix for a particular output channel layout(s) by means of the rendering exception matrix.

  If *m_ bAmbisonicEncObjFlag* is FALSE, then decoded waveforms represent generic audio tracks that shall be:

  - panned into a multichannel output bus if the *m_bObjectRendExceptPresent* is set to TRUE; or
  - exported to the system layer for further processing if the *m_bObjectRendExceptPresent* is set to FALSE.

- For objects using a channel mask, the rendering exceptions may be used to specify mix matrices for up to four explicit output layouts. For mono and stereo layouts (*unChLayoutIndex* < 2) down-mix matrices cannot be specified. Channel-based objects with *m_bObjectRendExceptPresent* equal to FALSE shall be mixed exclusively into the corresponding channel of the output mixing bus.

- For 3D objects, if *m_bObjectRendExceptPresent* is TRUE, then a rendering exception matrix for this 3D object is transmitted in the stream.

  Rendering exceptions for 3D objects allow a content creator to overwrite default renderer behavior for up to four explicit output layout(s). This is achieved by means of transmitting a custom mixing matrix (rendering exception matrix) for particular layout(s).

7.8.11.21 Object Properties with Multiple Updates Per Frame

Object properties that influence the movement of an object in 3D space may need to be updated more often than once per frame. These properties are described in this metadata block.

The metadata sampling period is every 512th tick of the clock whose rate is defined by *m_unClockRateInHz* (see clause 6.4.6.6). For each metadata sampling period the flag (*m_bPerSamplPeriodObjMDUpdFlag*) indicates if there is a metadata update at that sampling period.

Pseudo-code for initialization of this block of metadata is given in Table 7-33.

Pseudo-code for extraction of this block of metadata is given in Table 7-32.

Table 7-32: ExtractInvQuantMultiUpdtObjMD

Syntax	Reference

`void CL3DMD::ExtractInvQuantMultiUpdtObjMD(bool bObjStartFrameFlag, uint unObjectID,`
`  uint unFrameDuration, uint unOutChMask)`
`{`
`  uint bFirstSamplPointofFirstFrame;`
`  uint unPrevSamplPerIndex;`
`  uint unOutChMaskFullBWChOnly;`
 	
`  // Initialization`
`  unOutChMaskFullBWChOnly = InitMultiUpdtObjMD(unFrameDuration, unOutChMask);`
`  unPrevSamplPerIndex = m_unNumMDSamplPoints - 1;`
`  // Metadata sampling point loop`
`  for (uint nsmplp = 0; nsmplp < m_unNumMDSamplPoints; nsmplp++)`
`  {`
`  if (bObjStartFrameFlag == TRUE && nsmplp == 0)`
`  {`
`  m_bPerSamplPeriodObjMDUpdFlag = TRUE;`
`  bFirstSamplPointofFirstFrame = TRUE;`
`  }`
`  else {`
`  // For full channel-based mix (unObjectID=256) metadata update can only`
	7.8.11.22.1

Syntax	Reference

`// be TRUE at first sampling point (nsmplp=0)`
`m_bPerSamplPeriodObjMDUpdFlag= (unObjectID == 256) ? FALSE : (bool)ExtractBits(1);`
`bFirstSamplPointofFirstFrame = FALSE;`
`}`
	7.8.11.22.2

`// Copy metadata from the previous sampling instance`
`*m_ppObjGain[nsmplp] = *m_ppObjGain[unPrevSamplPerIndex];`
`if (m_bObject3DMetaDataPresent) {`
`// Copy source properties`
`for (uint unsrc = 0; unsrc < m_ucNum3DSourcesInObj; unsrc++) {`
`m_All3DSourcesInObj[unsrc*m_unNumMDSamplPoints + nsmplp] =`
`m_All3DSourcesInObj[unsrc*m_unNumMDSamplPoints + unPrevSamplPerIndex];`
`}`
`}`
`//`
`if (m_bObjectRendExceptPresent) {`
`// Copy rendering exceptions`
`*m_ppObjRendExcepMD[nsmplp] = *m_ppObjRendExcepMD[unPrevSamplPerIndex];`
`}`	

`if (m_bPerSamplPeriodObjMDUpdFlag) {`
`// Extract object gain parameter if update flag is TRUE.`
`uint unTemp = m_ppObjGain[nsmplp]->UpdateCode(bFirstSamplPointofFirstFrame, 6);`
`if (m_ppObjGain[nsmplp]->m_bUpdateFlag) {`
`// Inverse quantization by table look-up`
`unTemp = unScaleFactorsTable[unTemp];`
`m_ppObjGain[nsmplp]->SetParameter(unTemp);`
`}`
	7.8.11.22.3

`if (m_bObject3DMetaDataPresent) {`
`uint unBits2CodeSrcIndex;`
`if (m_bMonoObjWithMultipleSourcesFlag) {`
`// Number of 3D sources defined in this update instance.`
`// It allows for 4, 12, 28 or 60 sources in total.`
`// Prefix-code of length 1, 2, 3 and 3 is carried correspondingly for`
`// each of the values in uctable.`
`uchar uctable[4] = { 2, 3, 4, 5 };`
`m_ucNum3DSourcesInObj = (uchar)ExtractVarLenBitFields(uctable) + 1;`
`// Adjust the size of vector holding all instances of 3D source class`
`if (m_All3DSourcesInObj.size() < m_unNumMDSamplPoints*m_ucNum3DSourcesInObj) {`
`m_All3DSourcesInObj.resize(m_unNumMDSamplPoints*m_ucNum3DSourcesInObj,`
`C3DSource(m_pBitStream));`
`}`
`for (uint unsrc = 0; unsrc < m_All3DSourcesInObj.size(); unsrc++) {`
`m_All3DSourcesInObj[unsrc * m_unNumMDSamplPoints +`
`nsmplp].m_bSrcActiveFlag = FALSE;`
`}`
`// Get number of bits needed for coding a source index`
`unBits2CodeSrcIndex = GetBitsToRepresentuintValue(m_ucNum3DSourcesInObj);`
`}`
`else {`
`m_ucNum3DSourcesInObj = m_ucNumWaveFormsInObj;`
`unBits2CodeSrcIndex = 0;`
`}`
`// 3D Source Loop`
`uint unsrc_index;`
`for (uint unsrc = 0; unsrc < m_ucNum3DSourcesInObj; unsrc++) {`
`if (m_bMonoObjWithMultipleSourcesFlag) {`
`// Extract source index when the m_bMonoObjWithMultipleSourcesFlag is TRUE`
`unsrc_index = (uint)ExtractBits(unBits2CodeSrcIndex);`
`m_All3DSourcesInObj[unsrc_index*m_unNumMDSamplPoints + nsmplp].m_ucChIndex`
`= 0;`
`}`
`else {`
`unsrc_index = unsrc;`
`m_All3DSourcesInObj[unsrc_index*m_unNumMDSamplPoints + nsmplp].m_ucChIndex`
`= unsrc;`
`}`
`// Extract 3D source properties`
`m_All3DSourcesInObj[unsrc_index*m_unNumMDSamplPoints +`
`nsmplp].Extract3DSourceProperties(m_bChMaskObjectFlag, bFirstSamplPointofFirstFrame,`
`m_unObjTypeDescrIndex, m_bEnhancedREConfigParamsPresent);h`
`}`
`} // End of if (m_bObject3DMetaDataPresent)`
`if (m_bObjectRendExceptPresent) {`
`/ Extract the block of metadata corresponding to the rendering exceptions`
`m_ppObjRendExcepMD[nsmplp]->ExtractREObjectMD(bObjStartFrameFlag,`
`unOutChMaskFullBWChOnly);`
	7.8.11.22.4
	7.8.11.22.5
	7.8.11.27
	7.8.11.29

Syntax	Reference

}
} // End of m_bPerSamplPeriodObjMDUpdFlag is TRUE case
else {
// Case of m_bPerSamplPeriodObjMDUpdFlag is FALSE
// Reset the update flags for this instance
m_ppObjGain[nsmplp]->m_bUpdateFlag = false;
if (m_bObject3DMetaDataPresent) {
for (uint unsrc = 0; unsrc < m_ucNum3DSourcesInObj; unsrc++) {
m_All3DSourcesInObj[unsrc*m_unNumMDSamplPoints + nsmplp].ResetUpdateFlags();
}
}
if (m_bObjectRendExceptPresent) {
m_ppObjRendExcepMD[nsmplp]->m_bObjectRendExceptUpdateFlag = false;
}
} // End of m_bPerSamplPeriodObjMDUpdFlag is FALSE case
// Set previous update instance for the next iteration
unPrevSamplPerIndex = nsmplp;
} // End of metadata sampling points loop indexed by the value of nsmplp
} // End of CL3DMD::ExtractInvQuantMultiUpdtObjMD function	

7.8.11.22 Parameters for ExtractInvQuantMultiUpdtObjMD

7.8.11.22.1 Initialize Multiple Update Object Metadata

Function for initialization of Object Metadata that may be updated multiple times per frame, shown in Table 7-33, in particular creation of the 3D source class instances. The metadata sampling period is every 512-th clock tick whose rate is defined by *m_unClockRateInHz* (defined within FTOC). For each metadata sampling period the flag (*m_bPerSamplPeriodObjMDUpdFlag*) indicates if there is a metadata update for that sampling period.

Table 7-33: InitMultiUpdtObjMD

Syntax	Reference

uint CL3DMD::InitMultiUpdtObjMD(uint unFrameDuration, uint unOutChMask)
{
// The metadata sampling period is every 512-th tick of the sample clock
m_unNumMDSamplPoints = unFrameDuration >> 9; // unFrameDuration / 512
if (m_bObject3DMetaDataPresent) {
if (m_All3DSourcesInObj.size() < m_unNumMDSamplPoints*m_ucNumWaveFormsInObj) {
m_All3DSourcesInObj.resize(m_unNumMDSamplPoints*m_ucNumWaveFormsInObj,
C3DSource(m_pBitStream));
}
}
if (m_ppObjGain == NULL) {
m_ppObjGain = new CParamUpdSyncOrAnyFrame<uint> *[m_unNumMDSamplPoints];
for (uint unupd = 0; unupd < m_unNumMDSamplPoints; unupd++)
{
m_ppObjGain[unupd] = new CParamUpdSyncOrAnyFrame<uint>((1 << 15),
m_pBitStream); // Default gain is 1 in Q15 fixed point format
}
}
uchar ucNumFulBWWaveFormsInObj;
uchar unOutChMaskFullBWChOnly;
uchar ucNumFulBWWaveFormsInOutLayout;
 	
unOutChMaskFullBWChOnly = unOutChMask & (~(LFE1));
unOutChMaskFullBWChOnly = unOutChMaskFullBWChOnly & (~(LFE2));
ucNumFulBWWaveFormsInOutLayout = GetNumSpeakersInLayout(unOutChMaskFullBWChOnly);
	7.8.11.14.3

if (m_bObjectRendExceptPresent && (m_ppObjRendExcepMD == NULL)) {
uint unFullBWChMask;
if (m_bChMaskObjectFlag) {
unFullBWChMask = m_uint32ChActivityMask & (~(LFE1));
unFullBWChMask = unFullBWChMask & (~(LFE2));
ucNumFulBWWaveFormsInObj = GetNumSpeakersInLayout(unFullBWChMask);
	7.8.11.14.3

}
else {
unFullBWChMask = 0;
ucNumFulBWWaveFormsInObj = m_ucNumWaveFormsInObj;
}
m_ppObjRendExcepMD = new CRendExcepMD * [m_unNumMDSamplPoints];
for (uint unupd = 0; unupd < m_unNumMDSamplPoints; unupd++) {
m_ppObjRendExcepMD[unupd] = new CRendExcepMD(m_pBitStream,
ucNumFulBWWaveFormsInObj, ucNumFulBWWaveFormsInOutLayout, unFullBWChMask);	

Syntax	Reference

}
} // End of if (m_bObjectRendExceptPresent && (m_ppObjRendExcepMD == NULL))
return unOutChMaskFullBWChOnly;
} // End of InitMultiUpdtObjMD()	

7.8.11.22.2 m_bPerSamplPeriodObjMDUpdFlag

This 1-bit flag is extracted for each metadata sampling period. When *m_bPerSamplPeriodObjMDUpdFlag* = TRUE for a sampling period, there is a metadata update in this sampling period.

In the first sampling point (when *nsmplp*=0) of the object start frame (when *bObjStartFrameFlag* is TRUE), *m_bPerSamplPeriodObjMDUpdFlag* defaults to TRUE and is not transmitted in the bitstream.

For a full channel-based mix (*unObjectID*=256) this metadata chunk is transmitted only in frames where the *bObjStartFrameFlag* is TRUE. For this type of object the metadata update flag *m_bPerSamplPeriodObjMDUpdFlag* is by default set to FALSE for all sampling points where *nsmplp* > 0.

7.8.11.22.3 m_ppObjGain

This 6-bit field is present in the stream only if its corresponding update flag (*m_ppObjGain[nsmplp]->m_bUpdateFlag*) is set to TRUE. If update is present the new index is extracted within the *UpdateCode() function*. This index is used for the look-up into Scale Factor Table (clause 5.2.2.1). The extracted value represents the object gain that is to be applied, during playback, to all waveforms/sources that belong to this object.

7.8.11.22.4 m_ucNum3DSourcesInObj

This variable length field is transmitted only if *m_bObject3DMetaDataPresent* and *m_bMonoObjWithMultipleSourcesFlag* are both TRUE. It indicates the number of 3D sources that are defined for this object at the particular update instance. Values up to 4, 12, 28 or 60 are transmitted using either 3, 5, 7 or 8 bit variable length code.

When *m_bMonoObjWithMultipleSourcesFlag* is FALSE, *m_ucNum3DSourcesInObj* is equal to the number of waveforms *m_ucNumWaveFormsInObj*.

7.8.11.22.5 unsrc_index (3D Source Index)

This variable length field is transmitted only if *m_bObject3DMetaDataPresent* and *m_bMonoObjWithMultipleSourcesFlag* are both TRUE. It represents an index associated with a source within the list of stored object sources. During the life time of a 3D source, its associated source index is constant. The length of this field is calculated based on the number 3D sources (*m_ucNum3DSourcesInObj*).

In the case of a mono waveform with multiple sources (*m_bMonoObjWithMultipleSourcesFlag* is TRUE), since the sources can appear and disappear at every sampling instance, the source index is needed to track the existing sources in the list. When *m_bMonoObjWithMultipleSourcesFlag* is FALSE each source is associated with one waveform and all waveforms are persistent during the life of an object. Consequently, the 3D source index is not transmitted but rather deduced from the packing order.

7.8.11.23 m_rPerObjExpWinLambda

The fields related to exponential smoothing parameters may be present in the stream only if either *m_bObject3DMetaDataPresent* is TRUE or *m_bObjectRendExceptPresent* is TRUE.

An exponential window lambda parameter characterizes the smoothing of this object's contribution coefficients into the output bus. The range of *m_rPerObjExpWinLambda* parameter is from 0 to 0,9999 where 0 corresponds to the no smoothing case and 0,9999 corresponds to the smoothest case.

The *m_rPerObjExpWinLambda* class parameter is instantiated as a float with predefined value of LAMBDA_EXP_WINDOW_SMOOTHING.

If the *m_rPerObjExpWinLambda.m_bUpdateFlag* is TRUE, a 5-bit code is extracted from the stream. If the *m_rPerObjExpWinLambda.m_bUpdateFlag* is FALSE:

- If *bObjStartFrameFlag* is TRUE, the parameter is set by default to LAMBDA_EXP_WINDOW_SMOOTHING.
- If *bObjStartFrameFlag* is FALSE, the parameter maintains its value.

Extracted code is translated to the lambda parameter by look-up into the Inverse Quantization Table for the Exponential Window Smoothing Parameter Lambda (clause 5.2.2.7).

7.8.11.24 m_bObjecMDExtensionPresent

This 1-bit flag is transmitted in the stream in case when *unObjectID* < 256. In case when *unObjectID* = 256, *m_bObjecMDExtensionPresent* is by default set to FALSE. When *m_bObjecMDExtensionPresent* is TRUE, then a block of object metadata extension fields is transmitted in the stream.

Decoders shall be able to navigate past these extension fields.

7.8.11.25 Size of Object Metadata Extension Fields

This variable length field is transmitted in the stream if the *m_bObjecMDExtensionPresent* is TRUE. Variable code of length 8, 11, 14 or 16 bits allows the size of the extension fields to be up to 128, 640, 2 688 or 10 880 bits respectively.

7.8.11.26 Object Metadata Extension Fields

This block of data is present in the stream if the *m_bObjecMDExtensionPresent* is TRUE. Decoders shall navigate past these fields in order to parse the metadata for the next object.

7.8.11.27 Extraction of 3D Source Properties

This block of metadata is present in the stream only if *m_bObject3DMetaDataPresent* is TRUE. Each of the 3D sources defined for this object has corresponding properties. Extraction of parameters is listed in Table 7-34.

Table 7-34: Extract3DSourceProperties

Syntax	Reference

`void C3DSource::Extract3DSourceProperties(bool m_bChMaskObjectFlag,`
`bool bFirstSamplPointofFirstFrame,`
`uchar ucObjTypeDescriptorIndex,`
`bool bEnhancedREParamsPresent)`	

`{`	
`uint unTemp;`	
`int nTemp;`	
`if (m_bChMaskObjectFlag == FALSE)`	
`{`	
`if (bFirstSamplPointofFirstFrame && (ucObjTypeDescriptorIndex == ASSET_TYPE_COMPOSITE_MULTI_SRC)) {`	
`m_3DSrcTypeIndex = ExtractBits(3);`	
`}`	7.8.11.28.1
`// Extract 6-bit distance code if update flag is TRUE,`	
`unTemp = m_3DSrcRadius.UpdateCode(bFirstSamplPointofFirstFrame, 6);`	7.8.11.28.2
`if (m_3DSrcRadius.m_bUpdateFlag) {`	
`float rTemp;`	
`if (unTemp <= 32) {`	
`// Inside and on the unit sphere: linear proportionality of rTemp and unTemp`	
`//  rTemp is in range [0, 1/32, ... 1] with uniform step of 1/32`	
`rTemp = ((float)unTemp) * (1.0f / 32.f);`	
`}`	
`else {`	
`// Outside of the unit-sphere: quadratic relationship between rTemp and unTemp`	
`unTemp = unTemp - 32; // Translate to range 1 to 31`	
`//  rTemp is in range [1.0156, 1.0625, ... 16.0156] with non-uniform step`	
`rTemp = 1.0f + pow((float)unTemp * (1.0f / 8.0f), 2);`	
`}`	
`// Radius parameter is normalized to 1.0`	
`// The absolute distance may be obtained as:`	
`// m_ucRadiusRefernceUnitSphereInMeters * m_3DSrcRadius.GetParameter()`	

Syntax	Reference

m_3DSrcRadius.SetParameter(rTemp);
}
nTemp = (int)m_3DSrcAzimuth.UpdateCode(bFirstSamplPointofFirstFrame, 8);
	7.8.11.28.3

if (m_3DSrcAzimuth.m_bUpdateFlag) {
nTemp = -360 + 3 * nTemp;
nTemp = (nTemp == 219) ? 220 : nTemp;
nTemp = (nTemp == -219) ? -220 : nTemp;
nTemp = (nTemp>357) ? 357 : nTemp;
m_3DSrcAzimuth.SetParameter(nTemp);
}
// Extract 7-bit elevation angle code if update flag is TRUE
nTemp = (int)m_3DSrcElevation.UpdateCode(bFirstSamplPointofFirstFrame, 7);
	7.8.11.28.4

if (m_3DSrcElevation.m_bUpdateFlag) {
// The elevation angle of a source location in degrees in the range
// from -90 to 90 in steps of 1.5 degrees, stored in Q1 format
// as signed integers.
nTemp = -180 + 3 * nTemp;
nTemp = (nTemp>180) ? 180 : nTemp;
m_3DSrcElevation.SetParameter(nTemp);
}
if (bEnhancedREParamsPresent) {
unTemp = m_3DSrcRadiusAux.UpdateCode(bFirstSamplPointofFirstFrame, 6);
	7.8.11.28.5

}
else {
m_3DSrcRadiusAux.m_bUpdateFlag = FALSE;
}
 	
if (m_3DSrcRadiusAux.m_bUpdateFlag) {
float rTemp;
if (unTemp <= 32) {
// Inside and on the unit sphere: linear proportionality of rTemp and unTemp
// rTemp is in range [0, 1/32, ... 1] with uniform step of 1/32
rTemp = ((float)unTemp) * (1.0f / 32.f);
}
else {
// Outside of the unit-sphere: quadratic relationship between rTemp and unTemp
unTemp = unTemp - 32; // Translate to range 1 to 31
// rTemp is in range [1.0156, 1.0625, ... 16.0156] with non-uniform step
rTemp = 1.0f + pow((float)unTemp * (1.0f / 8.0f), 2);
}
}
else if (m_3DSrcRadiusAux.m_bParamPresent == FALSE) {
m_3DSrcRadiusAux.SetParameter(m_3DSrcRadius.GetParameter());
	7.8.11.28.5

}
if (bEnhancedREParamsPresent) {
unTemp = m_3DSrcSpreadParamOne.UpdateCode(bFirstSamplPointofFirstFrame, 6);
	7.8.11.28.6

if (m_3DSrcSpreadParamOne.m_bUpdateFlag) {
unTemp = (unTemp < 7) ? unTemp * 3 : (unTemp - 3) * 6;
m_3DSrcSpreadParamOne.SetParameter(unTemp);
}
unTemp = m_3DSrcSpreadParamTwo.UpdateCode(bFirstSamplPointofFirstFrame, 6);
	7.8.11.28.7

if (m_3DSrcSpreadParamTwo.m_bUpdateFlag) {
unTemp = (unTemp < 7) ? unTemp * 3 : (unTemp - 3) * 6;
m_3DSrcSpreadParamTwo.SetParameter(unTemp);
}
} // End of case when the bEnhancedREParamsPresent is TRUE
} // End of the m_bChMaskObjectFlag is FALSE case
// Indicate that current source is active
m_bSrcActiveFlag = true;
} // End of the Extract3DSourceProperties()	

7.8.11.28 Parameters for Extract3DSourceProperties

7.8.11.28.1 m_3DSrcTypeIndex

This parameter is present in the stream only for composite multi-source object type (ASSET_TYPE_COMPOSITE_MULTI_SRC) and only in the frames where the *bFirstSamplPointofFirstFrame* is TRUE. The 3-bit filed represents an index into a table of 3D renderer source types according to Table 7-35.

For all other object types the *m_3DSrcTypeIndex* is by default set to RENDERER_SOURCE_TYPE_DIRECT_SOUND however in this scenario the parameter *m_3DSrcTypeIndex* is ignored by the renderer.

Table 7-35: Renderer Source Types

Index	Renderer Source Types
0	RENDERER_SOURCE_TYPE_DIRECT_SOUND = 0
1	RENDERER_SOURCE_TYPE_REFLECTIONS_1
2	RENDERER_SOURCE_TYPE_REFLECTIONS_2
3	RENDERER_SOURCE_TYPE_REVERB
4	RENDERER_SOURCE_TYPE_RESERVED_1
5	RENDERER_SOURCE_TYPE_RESERVED_2
6	RENDERER_SOURCE_TYPE_RESERVED_3
7	RENDERER_SOURCE_TYPE_RESERVED_4

7.8.11.28.2 m_3DSrcRadius

A 3D source radius parameter represents the radial distance of a 3D source from the center of the coordinate system. The distance is normalized with respect to the unit sphere.

The *m_3DSrcRadius* class parameter is instantiated as float with predefined value of 1,0f.

If *m_bUpdateFlag* is TRUE, a 6-bit code is extracted from the stream. If *m_bUpdateFlag* is FALSE:

- the parameter is set by default to 1,0 if the *bFirstSamplPointofFirstFrame* is TRUE;
- the parameter maintains its value if the *bFirstSamplPointofFirstFrame* is FALSE.

Extracted code is translated to the radius parameter in the following manner:

- if the code is less or equal to 32 the radius parameter is in the range between 0,0 and 1,0 with uniform step of 1,0/32,0; in particular:

$$parameter = code/32{,}0$$

- if the code is greater than 32 the radius parameter has a quadratic relationship to its extracted code resulting in the parameter values in the range from 1,0156 to 16,0156 with a non-uniform step; in particular:

$$parameter = 1{,}0 + ((code - 32)/8{,}0)^2$$

To obtain the absolute value of the 3D source radius in meters, the normalized radius parameter is multiplied by the value of *m_ucRadiusRefernceUnitSphereInMeters* (available within the *C0x01MDChunkBody* class).

7.8.11.28.3 m_3DSrcAzimuth

The 3D source azimuth parameter represents an azimuth angle, expressed in degrees, of a 3D source position in polar coordinates.

The *m_3DSrcAzimuth* class parameter is instantiated as an integer without a predefined value.

If the *m_bParamPresent* and the *m_bUpdateFlag* are TRUE an 8-bit code is extracted from the stream. If the *m_bUpdateFlag* is FALSE:

- the parameter is set by default to 0 if *bFirstSamplPointofFirstFrame* is TRUE;
- the parameter maintains its value if *bFirstSamplPointofFirstFrame* is FALSE.

The azimuth parameter is in the range from -180 degrees to 178,5 degrees in steps of 1,5 degrees and is stored in Q1 format as a signed integer, in a range from -360 to 357. Extracted code is translated to the azimuth parameter in the following manner:

```
parameter =  - 360 + 3 * code
parameter = (parameter == 219) ? 220 : parameter
parameter = (parameter == -219) ? -220 : parameter
```

```
parameter = (parameter>357) ? 357 : parameter
```

where the rounding of ±219 to ±220 was done in order to guarantee that the angles of ±110 are represented exactly.

7.8.11.28.4 m_3DSrcElevation

A 3D source elevation parameter represents an elevation angle, expressed in degrees, of a 3D source position in polar coordinates.

The *m_3DSrcElevation* class parameter is instantiated as an integer with a predefined value of 0.

If *m_bUpdateFlag* is TRUE a 7-bit code is extracted from the stream. If *m_bUpdateFlag* is FALSE:

- the parameter is set by default to 0 if the *bFirstSamplPointofFirstFrame* is TRUE;
- the parameter maintains its value if the *bFirstSamplPointofFirstFrame* is FALSE.

The elevation parameter is in the range from -90 degrees to 90 degrees in steps of 1,5 degrees and is stored in Q1 format as a signed integer, in a range from -180 to 180. Extracted code is translated to the elevation parameter in the following manner:

```
parameter = -180 + 3 * code
parameter = (parameter>180) ? 180 : parameter
```

7.8.11.28.5 m_3DSrcRadiusAux

m_3DSrcRadiusAux is transmitted in the stream only if m*_bEnhancedREConfigParamsPresent* is TRUE. If a 3D source has a radial spread this is indicated by presence of *m_3DSrcRadiusAux* parameters.

When the *m_3DSrcRadiusAux.m_bParamPresent* is:

- TRUE, then a 3D source is radially spread between the radius value indicated by *m_3DSrcRadius* and the radius value indicated by *m_3DSrcRadiusAux.*
- FALSE, this indicates that a 3D source does not have a radial spread and it is positioned on the sphere with radius value indicated by *m_3DSrcRadius.* Furthermore the value of auxiliary radius parameter is by default set to the radius value indicated by *m_3DSrcRadius*.

If *m_3DSrcRadiusAux.m_bParamPresent* is TRUE the transmission and coding of *m_3DSrcRadiusAux* parameters in the bitstream is as described for the *m_3DSrcRadius*.

7.8.11.28.6 m_3DSrcSpreadParamOne

A 3D source spread *parameter one* represents a spreading angle in degrees that is interpreted differently depending on the value of *bUseDivergApperObjSpread*. In particular:

- When *bUseDivergApperObjSpread* = FALSE, *m_3DSrcSpreadParamOne* is an angle defining an azimuthal spread of a source symmetric around the 3D source position.
- When *bUseDivergApperObjSpread* = TRUE, *m_3DSrcSpreadParamOne* is an angle of the horizontal arc (latitude) centered at the 3D sound source position.

m_3DSrcSpreadParamOne is instantiated as an integer with a predefined value of 0.

If *m_bUpdateFlag* is TRUE, a 6-bit code is extracted from the stream. If *m_bUpdateFlag* is FALSE:

- If *bFirstSamplPointofFirstFrame* is TRUE, the parameter is set to 0.
- If *bFirstSamplPointofFirstFrame* is FALSE, the parameter maintains its value.

The values of *m_3DSrcSpreadParamOne* are in range from 0 to 360 degrees in steps of:

- 3 degrees when between 0 and 18 degrees; and
- 6 degrees when between 18 and 360 degrees.

Extracted code is translated to *m_3DSrcSpreadParamOne* by the following method:

```
parameter = (code < 7) ? code * 3 : (code - 3) * 6
```

7.8.11.28.7 m_3DSrcSpreadParamTwo

m_3DSrcSpreadParamTwo represents a spreading angle in degrees that is interpreted differently depending on the value of *bUseDivergApperObjSpread*. In particular:

- When *bUseDivergApperObjSpread* is FALSE, *m_3DSrcSpreadParamTwo* is an angle defining an elevation spread of a 3D source symmetric around the 3D source position.
- When *bUseDivergApperObjSpread* is TRUE, *m_3DSrcSpreadParamTwo* describes a spread of a source along the area on the surface of the sphere defined by the spherical cap that is symmetric around the 3D source position and with an aperture angle, in degrees, equal to the value of *m_3DSrcSpreadParamTwo*.

The *m_3DSrcSpreadParamOne* class parameter is instantiated as an integer with a predefined value of 0.

If *m_bUpdateFlag* is TRUE, a 6-bit code is extracted from the stream. If *m_bUpdateFlag* is FALSE:

- And if the *bFirstSamplPointofFirstFrame* is TRUE, the parameter is set by default to 0.
- And if the *bFirstSamplPointofFirstFrame* is FALSE, the parameter maintains its value.

The values of *m_3DSrcSpreadParamTwo* are in range from 0 to 360 degrees in steps of:

- 3 degrees when between 0 and 18 degrees; and
- 6 degrees when between 18 and 360 degrees.

Extracted code is translated to *m_3DSrcSpreadParamTwo* by the following equation:

```
parameter = (code < 7) ? code * 3 : (code - 3) * 6;
```

7.8.11.29 Extract Rendering Exception Object Metadata

The rendering exception metadata is transmitted only if *m_bObjectRendExceptPresent* is TRUE.

This metadata block, besides some control parameters, carries *m_nRECoeffsMatrix*, a matrix of mixing coefficients (mixing of object waveforms into a multichannel bus in specific channel layout) that has a different purpose depending on the object representation type:

- For channel mask based objects, *m_nRECoeffsMatrix* carries speaker remapping coefficients that allow the decoder to translate object's channel from its native channel layout to some different channel layout specified by control metadata; down-mixing is covered by this use case.
- For 3D-based objects, *m_nRECoeffsMatrix* carries rendering exception parameters that allow content creator to overwrite the default 3D renderer behavior for the output layouts specified by control metadata.
- For Ambisonic encoded objects, *m_nRECoeffsMatrix* represents a time domain Ambisonic decoding matrix for particular output channel layout as specified by control metadata. Note that this is applied directly to the decoded Ambisonic waveforms and in this case the Ambisonic adapter matrix (*m_pAmbiAdaptorMtrx*) is ignored.
- For audio tracks-based objects, *m_nRECoeffsMatrix* defines mixing of tracks into a particular channel layout, as specified by control metadata.

In order to maintain the mixing balance of all objects, the mixing matrix is constructed to preserve the energy of the object signals when mixing them into the output bus. However, in cases when multiple waveforms contribute to the same channel of the mixing bus, signal saturation may occur, so further attenuation of the mixing bus outputs may be needed. Saturation prevention is implemented by means of the presentation scaling parameters as described in clause 7.5.1.

Control metadata allows for transmission of up to 4 different rendering exception parameter sets each corresponding to a different channel layout of the mixing bus.

Note that any LFE channels are never covered or considered by this metadata.

Pseudo-code for extraction of this block of metadata is found in Table 7-36.

Table 7-36: ExtractREObjectMD

Syntax	Reference

`void CRendExcepMD::ExtractREObjectMD(bool bObjStartFrameFlag,`
`  uint unListeningLayoutMaskFullBWOnly)`
`{`
`  bool bExtendREtoSuperSetLayoutsFlag;`
`  bool bREApplicableFlag;`
`  uchar ucNumRESets;`
`  uint unREChannelMask;`
`  uint unNumChInRELayout;`
`  uint nuREMixBusChContribMask;`
`  uint unCounter, unIncr;`
`  int nTemp;`
`  int *pnCodes;`
`  int nSign;`
`  bool bContribCoeffsSigned;`
`  uchar uctable[4] = { 12, 16, 20, 32 };`	

`  bool UpdateFlag = (bObjStartFrameFlag) ? TRUE : (bool) ExtractBits(1);`	7.8.11.30.1
`  if (UpdateFlag) {`	
`    if (m_bObjFullBWChMask > 0) {`	
`      m_bUseREMForObjWithSameFullBWChMask = (ExtractBits(1) == 1) ? true : false;`	7.8.11.30.2
`    }`	
`    else {`	
`      m_bUseREMForObjWithSameFullBWChMask = false;`	
`    }`	
`    ucNumRESets = (uchar)ExtractBits(2) + 1;`	7.8.11.30.3
`    // This function is executed only if the RE update flag is TRUE hence`	
`    // reset the previous values`	
`    m_bREApplicableFlag = FALSE;`	
`    m_unNumChInRELayout = 0;`	
`    m_unREChannelMask = 0;`	
`    memset(m_nRECoeffsMatrix, 0,` `  sizeof(uint)*m_ucNumFulBWWaveFormsInObj* m_ucNumFulBWWaveFormsInOutLayout);`	
`    // Extract new RE parameters`	
`    for (uchar ncs = 0; ncs < ucNumRESets; ncs++){ // Loop over OW parameter sets`	
`      bContribCoeffsSigned = (ExtractBits(1) == 1) ? TRUE : FALSE;`	7.8.11.30.4
`      nSign = 1; // By default set sign to positive;`	
`      unREChannelMask = (uint)ExtractVarLenBitFields(uctable, FALSE);`	7.8.11.30.5
`      unNumChInRELayout = GetNumSpeakersInLayout(unREChannelMask);`	7.8.11.14.3
`      bExtendREtoSuperSetLayoutsFlag = (ExtractBits(1) == 1) ? TRUE : FALSE;`	7.8.11.30.6
`      // If bREApplicableFlag is TRUE the RE set is applicable to the output layout`	

`      bREApplicableFlag = CheckRESetApplicability(unREChannelMask,`
`  bExtendREtoSuperSetLayoutsFlag,`
`  unListeningLayoutMaskFullBWOnly);`
	7.8.11.30.7

`      if (bREApplicableFlag)`
`  if (m_ bREApplicableFlag == FALSE || (unNumChInRELayout > m_unNumChInRELayout)`
`  && m_bREApplicableFlag) {`
`  bSaveFlag = TRUE;`
`  }`
`  }`	

`      else {`
`  if (m_bREApplicableFlag == false && (unNumChInRELayout > m_unNumChInRELayout) )`
`  {`
`  bSaveFlag = true;`
`  }`
`  }`	

`      if (bSaveFlag) {`
`  unIncr = 1;`
`  pnCodes = m_nRECoeffsMatrix;`
`  m_unNumChInRELayout = unNumChInRELayout;`
`  m_unREChannelMask = unREChannelMask;`
`  m_bREApplicableFlag = bREApplicableFlag;`
`  }`	

`      else {`
`  unIncr = 0;`
`  pnCodes = &nTemp;`
`  }`	

`      unCounter = 0;`	
`      for (uchar nwf = 0; nwf < m_ucNumFulBWWaveFormsInObj; nwf++){`	
`        nuREMixBusChContribMask = (uint)ExtractBits(unNumChInRELayout);`	7.8.11.30.8
`        for (uint nn = 0; nn < unNumChInRELayout; nn++){` `  if ((nuREMixBusChContribMask >> nn) & 0x01){`	

Syntax	Reference

if (bContribCoeffsSigned) {
// If this is a signed contribution matrix then extract the sign bit
nSign = (int)((ExtractBits(1) == 1) ? -1 : 1);
}
// Extract 6-bit index corresponding to unsigned scale factor
nTemp = (int)ExtractBits(6);
} // End of: if ((nuREMixBusChContribMask >>nn) ...
else{
nTemp = 0; // 0 -> no contribution
} // End of else part of if ((nuREMixBusChContribMask [nwf]>>nn) ...
pnCodes[unCounter] = nSign * ((int)unScaleFactorsTable[nTemp]);
unCounter = unCounter + unIncr;
} // End of channels in the RE layout loop: for (nn=0; nn ...
} // End of waveform loop for (nwf=0; nwf<m_ucNumFulBWWaveFormsInObj; ncs++)
} // End of rendering exception set loop for (ncs=0; ncs<ucNumRESets; ncs++)
} // End of if (UpdateFlag)
} // End of CRendExcepMD::ExtractREObjectMD
	7.8.11.30.9

7.8.11.30 Parameters for ExtractREObjectMD

7.8.11.30.1 bObjectRendExceptUpdateFlag

If *bObjectRendExceptUpdateFlag* is FALSE, then the rendering exception metadata is not transmitted and maintains the value previously set. TRUE indicates that the rendering exception metadata for this object is transmitted in the current metadata sampling period.

7.8.11.30.2 m_bUseREMForObjWithSameFullBWChMask

m_bUseREMForObjWithSameFullBWChMask is transmitted in the stream only for channel-based objects (*m_bObjFullBWChMask* > 0). When *m_bUseREMForObjWithSameFullBWChMask* is TRUE, the rendering exception matrix (REM) applicable to the desired layout shall be applied to all channel mask-based objects that have the same full-bandwidth channel mask and do not have their own REM for the applicable output layout.

It is the encoder's responsibility to guarantee that, within the same audio presentation, there is at most one object with *m_bUseREMForObjWithSameFullBWChMask* set to TRUE for a given *m_bObjFullBWChMask*. If, within the same presentation, two or more channel-based active objects with the same *m_bObjFullBWChMask* have *m_bUseREMForObjWithSameFullBWChMask* set to TRUE then all REM's extracted within these objects shall be ignored.

For non-channel-based objects (*m_bObjFullBWChMask*=0) this flag is not transmitted and is set to FALSE.

7.8.11.30.3 ucNumRESets

This 2-bit field indicates the number of rendering exception (RE) parameter sets. The valid range is from 1 to 4.

7.8.11.30.4 bContribCoeffsSigned

If *bContribCoeffsSigned* is TRUE, then the mixing matrix has signed entries. If *bContribCoeffsSigned* is FALSE, then the entries of the mixing matrix are unsigned.

7.8.11.30.5 unREChannelMask

This variable length field indicates the channel layout bit-mask corresponding to the mixing bus of the current rendering exception parameter set. Variable length codes of 13, 18, 23 or 35 bits are used to extract a channel mask of up to 12, 16, 20 or 32 bits.

The number of channels (*unNumChInRELayout*) is calculated from *unREChannelMask*.

7.8.11.30.6 bExtendREtoSuperSetLayoutsFlag

If *bExtendREtoSuperSetLayoutsFlag* is TRUE, then the RE parameter set is applicable to any listening layout that includes all channels (not including LFE), defined by *unREChannelMask* and any arbitrary number of additional channels. If *bExtendREtoSuperSetLayoutsFlag* is FALSE, then the RE parameter set is applicable only to the listening layout that is defined by *unREChannelMask*. This logic is implemented in clause 7.8.11.30.7.

7.8.11.30.7 Checking Applicability of the RE Parameter Set

The pseudo-code in Table 7-37 demonstrates applicability of the RE parameter set. If *bExtendREtoSuperSetLayoutsFlag* is TRUE, the RE parameter set is applicable to any listening layout that includes all channels defined by the *unREChannelMask* and any arbitrary number of additional channels, (excepting the LFE). If *bExtendREtoSuperSetLayoutsFlag* is FALSE, then RE is applies only to the listening layout that is defined by *unREChannelMask*.

Table 7-37: CheckRESetApplicability

Syntax

```
Function for Checking Applicability of the RE Parameter Set

bool CRendExcepMD::CheckRESetApplicability(uint unREChannelMask,
                                           bool bExtendREtoSuperSetLayoutsFlag,
                                           uint unOutChMask)
{
  bool bApplicableFlag;

  if (bExtendREtoSuperSetLayoutsFlag) {
    bApplicableFlag = (bool) ((unREChannelMask & unOutChMask) == unREChannelMask);
  } else {
    bApplicableFlag = (bool) (unREChannelMask == unOutChMask);
  }
  return bApplicableFlag;
}
```

7.8.11.30.8 nuREMixBusChContribMask

This *unNumChInRELayout* bit mask indicates the channels of the RE mixing bus to which the current waveform is mixed in. Only the channels corresponding to "1" bits receive contribution and all channels corresponding to "0" bits by default receive zero contribution.

7.8.11.30.9 m_nRECoeffsMatrix[]

Contribution coefficients are transmitted only for channels corresponding to "1" bits within the *nuREMixBusChContribMask*.

Each coefficient is extracted in one or two parts depending on the value of *bContribCoeffsSigned*:

- Sign. When *bContribCoeffsSigned* is TRUE: 1-bit field is extracted indicating the sign where 1 implies negative sign (*sign_value* = -1) and 0 implies positive sign (sign_value = +1). When the *bContribCoeffsSigned* is FALSE: *sign_value* = +1 by default.

- 6-bit field representing an index (*unIndex*) for look-up into a table with scale factors represented in unsigned Q0.15 fixed point format as shown in Scale Factor Table (clause 5.2.2.1).

Entries of the *m_nRECoeffsMatrix* are signed integers in Q0.15 fixed point format. The values extracted from the scale table are translated to signed Q0.15 fixed point format as *m_nRECoeffsMatrix*[n] = sign_value * *unScaleFactorsTable*[unIndex]. *unScaleFactorsTable* is found in clause 5.2.2.1.

Within *m_nRECoeffsMatrix* the channels of the mixing bus are ordered according to *unREChannelMask* where channel(s) corresponding to the first "1" bit within the mask (starting from the LSB of the mask) is (are) first. In case where the "1" bit corresponds to the channel pair, according to Table 7-28 the left channel of the pair is counted first in the list.

7.8.11.30.10 Metadata Packet Payload

A packet payload extracted in current frame shall be placed into the appropriate location within the static metadata buffer *m_punStaticLDMetadata[]*, and in particular locations between *m_unStaticMDPacketIndex*, *m_unStaticMDPacketByteSize* and (*m_unStaticMDPacketIndex* + 1) * *m_unStaticMDPacketByteSize* - 1.

For the syntax of metadata captured in the *m_punStaticLDMetadata[]* refer to Static Loudness and Dynamics Metadata, (clause 7.7.5).

8 ACE for DTS-UHD

8.1 Overview

The ACE Decoder operates on ACE stereo streams, ACE mono streams and ACE LFE streams. Within an ACE frame, the streams have been separated into one of these three classifications. Stereo streams have an additional optimization that takes advantage of the information common to both channels.

The block diagram in Figure 8-1 shows the flow of data through the decoder for full bandwidth channels.

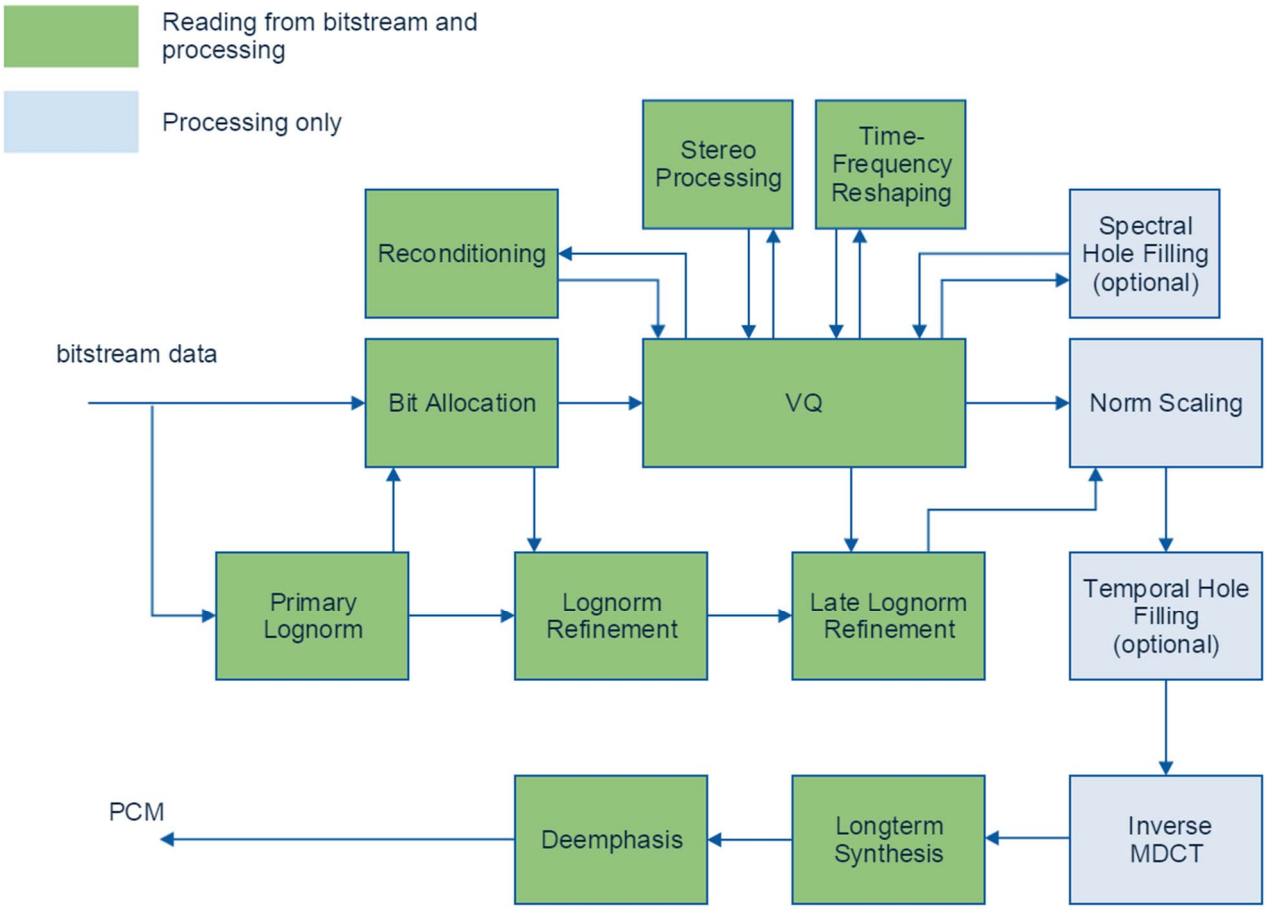

Figure 8-1: ACE Decoder

Decoding a mono or stereo ACE stream involves the following stages (see also Figure 8-1).

Primary Lognorm: a first approximation of spectral band power is retrieved from the bitstream. These power values are represented in the logarithmic domain, and referred to as lognorm values.

Bit Allocation: for each spectral band, the number of bits available for normalized spectral band decoding is computed. This computation depends on information in the bitstream and (potentially) primary lognorm values.

Lognorm Refinement: a small part of the bit allocation is used to refine the primary lognorm values.

VQ (Vector Quantization): the majority of the bit allocation is used to reconstruct normalized spectral frequency values, one spectral band at a time.

Reconditioning: for originally "full" spectral bands and small bit allocation values, the initially reconstructed sparse bands are reconditioned to better approximate a full spectrum.

Stereo processing: for stereo streams, the two reconstructed channels are transformed to an original (unmixed) representation.

Time-Frequency Reshaping: the spectral bands are pre-processed by a simple Haar transform, bringing all bands to the same time-frequency representation.

Spectral Hole Filling: bands with original energy that are reconstructed to all-zeros are then filled with 'noise', generated from earlier reconstructed band values.

Late Lognorm Refinement: any bits that have been left over after VQ are used to do a final lognorm refinement.

Norm Scaling: spectral bands are restored to the power values corresponding to the final lognorm values.

Temporal Hole Filling: all-zero spectral bands are potentially filled with synthetic noise depending on the spectral band power in previous frames.

Inverse MDCT: the spectral representation is transformed in a time-domain representation (PCM).

Longterm Synthesis: a long term synthesis filter is used to reconstruct signal components that have been removed at encoding time.

The remainder of clause 8 describes the bitstream extraction of the ACE decoder. Clause 9 describes the various methods involved in reconstructing the Linear PCM audio.

8.2 Fundamental Bitstream Operations for ACE

8.2.1 Align

Aligns the read pointer to a n-byte boundary.

Table 8-1: Align

Syntax

`// Read count: 0 ... 8*n-1 bits`
`void align(uint n);`
`void align() {`
`  align(1);`
`}`

Parameters

- **n** - Byte boundary to align to.

8.2.2 Skip

Advances the read pointer by n bytes. Byte alignment is implied.

Table 8-2: Skip

Syntax
`// Read count: n bytes` `void skip(uint n);`

Parameters

- **n** - Number of bytes to skip.

8.2.3 BitsUsed

BitsUsed(void) reports the position of the bitstream read pointer with respect to the origin (start point) of the bitstream. *BitsUsed(uint n)* set the current position of the bitstream read pointer to the value *n*. A typical use is the call *BitsUsed(0)*, sets the current position of the bitstream pointer as the origin of the bitstream.

Table 8-3: BitsUsed

Syntax

`// Report position of bitstream read pointer`
`int BitsUsed();`
 	
`// Set current bitstream read pointer to specified value n`
`int BitsUsed(int n)`

Parameters

- **n** - The new value of the current bitstream read pointer.

8.2.4 BitsLeft

Bitsleft(void) reports the number of bits left in the bitstream, that is, the number of bits that can be read before exceeding bitstream capacity.

Table 8-4: BitsLeft

Syntax
`// Report the number of bits left in the bitstream.` `int BitsLeft();`

8.2.5 BitsTotal

BitsTotal(void) reports the capacity of the bitstream. *BitsTotal(uint n)* sets the capacity of the bitstream to the value *n*.

Table 8-5: BitsTotal

Syntax

`// Report the capacity of the bitstream`
`int BitsUsed();`
 	
`// Set the capacity of the bitstream`
`int BitsTotal(int n)`

Parameters

- **n** - The new value of the bitstream capacity.

8.2.6 ReadBit

Readbit(void) extracts a single bit from the bitstream and returns a value *0* or *1*.

Table 8-6: ReadBit

Syntax

`// Read count: 1 bit`
 	
`uint ReadBit() {`
`  return (uint)ExtractBit(1);`
`}`

8.2.7 ReadBool

ReadBool(void) extracts a single bit from a bitstream and returns a value of either *FALSE* or *TRUE*.

Table 8-7: ReadBool

Syntax	Reference

`// Read count: 1 bit`
 	
`bool ReadBool() {`
`  return (bool)ReadBit();`
`}`
	8.2.3

8.2.8 ReadUint

ReadUint (uint *num_bits*) extracts *num_bits* from a bitstream to form an unsigned integer.

Table 8-8: ReadUint

Syntax	Reference

`// Read count: num_bits bits`
`uint ReadUint(uint num_bits) {`
`  uint result = 0;`
`  for (int i=0;i<num_bits;i++) {`
`  result = (result << 1) | ReadBit();`
`  }`
`  return result;`
`}`
	8.2.3

8.2.9 ReadInt

ReadInt (uint *num_bits*) extracts a 2's complement integer from a bitstream.

Table 8-9: ReadInt

Syntax	Reference

`// Read count:  num_bits bits`
`int ReadInt(uint num_bits) {`
`  if (num_bits == 0) { return 0; }`
`  int result = (int) ReadUint(num_bits);`
`  if (result >= ((1<<num_bits)>>1) {`
`  return (result - (1<<num_bits));`
`  }`
`  return result;`
`}`
	8.2.8

Parameters

- **num_bits** - The resolution of the signed integer.

8.2.10 ReadUnary

ReadUnary(*alpha*) extracts an unsigned integer *n* from the bitstream, with $0 <= n < alpha$. The value $n < alpha - 1$ is encoded as a sequence of *n* zeros, terminated by the symbol 1. The value $alpha - 1$ is encoded by a sequence of $alpha - 1$ zeros. *ReadUnary*(*alpha*) returns the value 0 for $alpha = 0,1$.

Table 8-10: ReadUnary

Syntax	Reference

`// read count: < alpha bits`
`uint ReadUnary(uint alpha) {`
`  if (alpha <= 1) {`
`  return 0;`
`  }`
`  uint result = 0;`
`  while ( result+1 < alpha ) {`
`  bool bit = ReadBit();`
`  if ( bit == 1)`
`  break;`
`  ++result;`
`  }`
`  return result;`
`}`
	8.2.3

Parameters

- **alpha** - The number of elements in the alphabet.

8.2.11 ReadUniform

ReadUniform(*alpha*) extracts an unsigned integer *n* from the bitstream, where $0 \leq n < alpha$.

The value n is encoded as an m-bit or (m-1)-bit integer, where $2^{(m-1)} < alpha \leq 2^m$.

When *alpha* is a power of two, *ReadUniform*(*alpha*) is equivalent to *ReadUint*(*m*).

Table 8-11: ReadUniform

Syntax	Reference

`// Read count: NumBits(alpha-1)-1 or NumBits(alpha-1), where NumBits(alpha-1) = m`
 	
`uint ReadUniform(uint alpha) {`
`  if (alpha <= 1) {`
`  return 0;`
`  }`
`  uint result = 0;`
`  uint m = NumBits(alpha-1);`
`  uint gap = (1<<m) - alpha;`
`  result = ReadUint(m-1);`
`  if (result >= gap) {`
`  result = (result << 1) | ReadBit();`
`  result -= gap;`
`  }`
`  return result;`
	8.3.1
	8.2.8
	8.2.3

Parameters

- **alpha** - The number of elements in the alphabet.

8.2.12 ReadGolomb

ReadGolomb(uint alpha, uint kparam) extracts an unsigned integer from the bitstream with the *kparam* least significant bits uniformly encoded, and the remaining most significant bits unary encoded.

Table 8-12: ReadGolomb

Syntax	Reference
`// Read count: variable`	
`uint ReadGolomb(uint alpha, uint kparam) {`	
`  if (alpha <= 1) { return 0; }`	
`  uint kmax = NumBits(alpha-1) - 1;`	8.3.1
`  if (kparam >= kmax) {`	
`    return ReadUniform(alpha);`	8.2.11
`  }`	
`  // Handle default case`	
`  uint result = ReadUint(kparam);`	8.2.8
`  result |= ReadUnary(1+((alpha-1)>>kparam)) << kparam;`	8.2.10
`  // Return`	
`  return result;`	
`}`	

Parameters

- **alpha** - The number of elements in the alphabet.
- **kparam** - The resolution of the binary part of the encoded value.

8.2.13 ReadGolombLimited

ReadGolombLimited(uint alpha, uint kparam, uint limit) extracts an unsigned integer from the bitstream. Values smaller than *limit* are Golomb-encoded with binary resolution *kparam*. Values larger than or equal to *limit* are encoded by first Golomb-encoding *limit* and then encoding the difference between the value and *limit* uniformly.

Table 8-13: ReadGolombLimited

Syntax	Reference
`// Read count: variable`	
`uint ReadGolombLimited(uint alpha, uint kparam, uint limit) {`	
`  if (alpha <= 1) { return 0; }`	
`  // Handle small limit`	
`  if (limit < (1 << kparam)) {`	
`    limit = 1 << kparam;`	
`  }`	
`  // Handle large limit`	
`  if (limit >= alpha) {`	
`    limit = alpha-1;`	
`  }`	
`  // Read with limited alpha`	
`  uint result = ReadGolomb(limit+1,kparam);`	8.2.12
`  // Handle values greater than or equal to limit`	
`  if (result == limit) {`	
`    return ReadUniform(alpha - limit) + limit;`	8.2.11
`  }`	
`  return result;`	
`}`	

Parameters

- **alpha** - The number of elements in the alphabet.
- **kparam** - The resolution of the binary part of the encoded value.
- **limit** - The maximum value of the Golomb encoded values.

8.2.14 ReadGolombWithParams

ReadGolombWithParams(GolombParams gparams) extracts an unsigned integer from the bitstream using *ReadGolombLimited*, with its parameters aggregated in a *GolombParams* datatype.

Table 8-14: ReadGolombWithParams

Syntax	Reference

`// Aggregate for Golomb parameters`
`typedef struct {`
`  uint alpha;`
`  uint kparam;`
`  uint limit;`
`} GolombParams;`
 	
`// Read count: variable`
`uint ReadGolombWithParams(GolombParams gparams) {`
`  return ReadGolomb(gparams.alpha,gparams.kparam, gparams.limit);`
`}`
	8.2.12

8.2.15 ReadVLC

ReadVLC(primary_table, escape_table) extracts an unsigned integer from the bitstream using *ReadUint*(), with the resolution of the *ReadUint* call read from the bitstream using *ReadUnary*. The value returned is recursively incremented by the alphabet size of previous unused resolutions. The parameter list of *ReadVLC* allows for escape values. If the value is large and falls within the escape range, the value returned is obtained by calling *ReadUint* with a dedicated escape resolution table (return values recursively incremented by previous unused resolutions).

ReadVLC () is said to be operating in primary mode when the escape mode is not triggered. Otherwise, it is said to be operating in escape mode.

Table 8-15: ReadVLC

Syntax	Reference

`// Table structure + associated methods`
`typedef struct {`
`  uint size;`
`  uint list[];`
`  uint size() { return size; }`
`  uint operator[](uint index) { return list[index]; };`
`  uint alpha(uint index) {`
`  uint result = 0;`
`  for (int i=0;i<index;i++) {result += (1u<<list[i]); }`
`  return result;`
`  }`
`  uint alpha() { return alpha(size); }`
`} Table;`	

`// Bit count: variable` `uint ReadVLC(Table primary_table, Table escape_table) {`	8.2.15
`  uint result = 0;`	
`  uint primary = ReadUnary(primary_table.size());`	8.2.10
`  uint result = ReadUint(primary_table[primary]) + primary_table.alpha(primary);`	8.2.8
`  if (result + escape_table.size() >= primary_table.alpha()) { // escape mode`	
`  uint escape = result + escape_table.size() - primary_table.alpha();`	
`  result = ReadUint(escape_table[escape]) + escape_table.alpha(escape);`	8.2.8
`  result += primary_table.alpha() ;`	
`  }`	
`  return result;` `}`	

Parameters

- **primary_table** - The table of resolutions for primary mode.
- **escape_table** - The table of resolutions for escape mode.

8.2.16 ReadLimitsVLC

ReadLimitsVLC(res_table) extracts an unsigned integer from the bitstream using a sequence of resolutions given by *res_table*. The sequence stops when the partial value read from the bitstream is smaller than the maximum value for the current resolution.

Table 8-16: ReadLimitsVLC

Syntax	Reference

uint ReadLimitsVLC(Table res_table) {
  uint result = 0;
  for (int i=0;i<res_table.size();i++) {
  uint value = ReadUint(res_table[i]);
  result += value;
  if ((value+1) < (1 << res_table[i])) { // no escape
  break;
  }
  }
  return result;
}
 	
const Table FrameTable = {1,1,2,2,2,4,4,16};
uint ReadLimitsVLC() {
  return ReadLimitsVLC(FrameTable);
}
	8.2.8

Parameters

- **res_table** - The table of resolutions.

8.2.17 ReadNonUniformFiveTen

ReadNonUniformFiveTen() extracts an unsigned integer from the bitstream. Values smaller than 24 are represented by 5 bits. Values larger than 23 and smaller than 280 are represented by 10 bits. Values larger than 279 and smaller than 280 + (1 << 22) are represented by 32 bits.

Table 8-17: ReadNonUniformFiveTen

Syntax	Reference

uint ReadNonUniformFiveTen()
{
  uint result = 0;
  result += ReadUint(5);
  if (result > 23) {
  result = ((result - 24)<<5) + ReadUint(5);
  }
  if (result == 279) {
  result += ReadUint(22);
  }
  return result;
}
	
	8.2.8
	8.2.8
	
	8.2.8

8.3 Auxiliary Functions

8.3.1 NumBits

NumBits(*value*) returns the number of minimal number bits required to represent *value*.

Table 8-18: NumBits

Syntax

uint NumBits(uint value) {
  uint result = 0;
  while (value != 0) {
  value >>= 1;
  result++;
  }
  return result;
}

Parameter

- **value** - The value for which the number of bits in its representation is returned.

8.3.2 Modular

Modular(*value*,*low*,*high*) computes *value* (mod *alpha*) represented between *low* (inclusive) and *high* (inclusive), where $alpha = high - low + 1$.

Table 8-19: Modular

Syntax

int Modular (int value, int low, int high) {
  ASSERT(low <= high);
  uint alpha = high - low + 1;
  int result = value;
  while (result > high) { result -= alpha; }
  while (result < low) { result += alpha; }
  return result;
}

Parameters

- **value** - The value for which the modular value is to be computed.
- **low** - The lower bound of the representation interval.
- **high** - The upper bound of the representation interval.

8.3.3 NegativeRiceMap

NegativeRiceMapDecode(*value*) decodes a signed from an unsigned integer representation. It differs from *PositiveRiceMapDecode*(*value*) by a sign change.

Table 8-20: NegativeRiceMapDecode

Syntax

int NegativeRiceMapDecode(uint value) {
 	
  if (value & 1) {
  return -(int)(value+1)/2;
  else {
  return (int)value/2;
  }
}

Parameter

- **value** - The unsigned integer representation of a signed integer.

8.3.4 PositiveRiceMap

PositiveRiceMapDecode(*value*) decodes a signed from an unsigned integer representation. It differs from *NegativeRiceMapDecode*(*value*) by a sign change.

Table 8-21: PositiveRiceMapDecode

Syntax

int PositiveRiceMapDecode(uint value) {
  return -NegativeRiceMapDecode(value);
}

Parameter

- **value** - The unsigned integer representation of a signed integer.

8.3.5 CenterMap

CenterMap is an efficient method to encode a non-negative integer concentrated around a center value *c*, by ordering unsigned integers in a finite alphabet in increasing order of distance to *c*. Given an alphabet size *alpha* and a center *c* in the range $0 < c < alpha$ and assuming $c \leq \lfloor (alpha - 1)/2 \rfloor$ the map *EncodeCenterMap* is defined as follows:

```
c -> 0

c+1 -> 1

c-1 -> 2

...

2c -> 2c-1

0 -> 2c

2c+1 -> 2c+1

alpha-1 -> alpha-1
```

For $c > \lfloor (alpha - 1)/2 \rfloor$, the map is the mirror image of the previous case. *CenterMapDecode*(*alpha*, *c*, *y*) recovers *x* from its mapped value *y*.

Table 8-22: DecodeCenterMap

Syntax

uint DecodeCenterMap(uint alpha, uint c, uint y) {
  ASSERT(c < alpha && y < alpha);
  if (c <= (alpha-1)/2) {
  if (y > 2*c) {
  return y;
  }
  if (y%2 == 0) {
  return c - y/2;
  }
  else {
  return c + (y+1)/2;
  }
  }
  else {
  return alpha - 1 - DecodeCenterMap(alpha, alpha-1-c, y);
  }
}

Parameter

- **alpha** - The size of the alphabet, defining the input range between 0 (inclusive) and alpha (exclusive).
- **c** - The center value from which distances are taken.
- **y** - The mapped input value.

8.3.6 Rand

The function *Rand(float lb, float ub)* draws with uniform probability a floating point value in the interval defined by its lower bound (inclusive) and its upperbound (exclusive).

Table 8-23: Rand

Syntax

```
// Pseudo-random number generator
//
// parameter[in] lb: lowerbound of interval (inclusive)
// parameter[in] ub: upperbound of interval (exclsuive)
// return: floating value in defined interval

float Rand(float lb, float ub);
```

Parameter

- **lb** - Lower bound (inclusive) of selection interval.
- **ub** - Upper bound (exclusive) of selection interval.

8.4 ACE Data Types

8.4.1 FixedPoint

An instance of class *FixedPoint* represents the number *_data / (1 << PRECISION)*, where *_data* is the internal data carrier, and *PRECISION* is the fixed point precision parameter.

Table 8-24: FixedPoint

Syntax

```
// FixedPoint Class (datatype)

template<int PRECISION>
class FixedPoint {

  // Initialization from integer and fractional part
  FixedPoint(int intpart, int fracpart) {
    _data = (intpart << PRECISION) + fracpart;
  }

  // Initialization from integer
  FixedPoint(int intpart) {
    _data = (intpart << PRECISION);
  }

  // Initialization from float
  FixedPoint(float f) {
    _data = floor(f * (1 << PRECISION) + 0.5);
  }

  // Initialization from FixedPoint
  FixedPoint(FixedPoint &fp) {
    _data = fp.data;
  }

  // Raw data retrieval
  int data() {
    return _data;
  }

  // Raw data setting
  int data(int d) {
    _data = d;
    return _data;
  }

  // Return a fraction of the form numer * 2^(-logdenom), to a precision of PRECISION
  static FixedPoint Fraction(int numer, int logdenom) {
    return (logdenom > PRECISION) ?
      FixedPoint(0, numer >> (logdenom-PRECISION)) :
      FixedPoint(0, numer << (PRECISION-logdenom));
  }

  // conversion to integer
```

Syntax

```
operator int() { return _data >> PRECISION; }

// conversion to float
operator float() {return _data / (float) (1 << PRECISION); }

// rounding to nearest integer
int round() {
  return (_data + (((1 << PRECISION)>>1) - 1)) >> PRECISION;
}

// assignment from integer
FixedPoint &operator=(int d) {
  _data = (d<<PRECISION);
  return *this;
}

// assignment from float
FixedPoint &operator=(float f) {
  _data = floor(f * (1 << PRECISION) + 0.5);
  return *this;
}

// Equality test
bool operator==(const FixedPoint &fp) const {
  return _data == fp._data;
}

// Test equality to an int
bool operator==(int y) const {
  return (*this) == FixedPoint(y);
}

//Test if zero
bool IsZero() const {
  return _data == 0;
}

//Inequality test
bool operator!=(const FixedPoint &fp) const {
  return _data != fp._data;
}

//Less than or equal
bool operator<=(const FixedPoint &fp) const {
  return _data <= fp._data;
}

//Less than
bool operator<(const FixedPoint &fp) const {
  return _data < fp._data;
}

//Greater than or equal
bool operator>=(const FixedPoint &fp) const {
  return _data >= fp._data;
}

// Greater than
bool operator>(const FixedPoint &fp) const {
  return _data > fp._data;
 }

// Addition assignment
FixedPoint &operator+=(const FixedPoint &fp) {
  _data += fp._data;
  return *this;
}

// Addition of int and assignment
FixedPoint &operator+=(int u) {
  FixedPoint fp(u);
  _data += fp._data;
  return *this;
}

//Subtraction assignment
FixedPoint &operator-=(const FixedPoint &fp) {
```

Syntax

```
  _data -= fp._data;
  return *this;
}

//Subtraction of int and assignment
FixedPoint &operator-=(int u) {
  FixedPoint fp(u);
  _data -= fp._data;
  return *this;
}

// Multiplication assignment
FixedPoint &operator*=(const FixedPoint &y) {
  _data = (_data  * y._data) >> PRECISION;
  return *this;
}

// Multiplication by int and assignment
FixedPoint &operator*=(int y) {
  _data *= y;
  return *this;
}

// Division assignment
FixedPoint &operator/=(const FixedPoint &y) {
  _data = (_data << PRECISION) / y._data;
  return *this;
}

// Division by int and assignment
FixedPoint &operator/=(int y) {
  _data /= y;
  return *this;
}

// Right shift
FixedPoint &operator>>=(unsigned int i) {
  _data >>= i;
  return *this;
}

// Left shift
FixedPoint &operator<<=(unsigned int i) {
  _data <<= i;
  return *this;
}

// IntPart
int IntPart() {
  return _data >> PRECISION;
}

// FracPart
int FracPart() {
  return _data & ((1<<PRECISION) - 1);
}

// number of bits used to index exponential_table, i.e.
// the logarithm of exponential_table_size
const int EXPONENTIAL_TABLE_RESOLUTION = 7;

// See documentation of the static class member exponential_table
const int EXPONENTIAL_TABLE_SIZE = 1 << exponential_table_resolution;

// A table containing a fixed point representation of 2^{i / EXPONENTIAL_TABLE_SIZE}
// for 0 <= i < exponential_table_size
const uint EXPONENTIAL_TABLE[] = {
  512, 515, 518, 520, 523, 526, 529, 532,
  535, 538, 540, 543, 546, 549, 552, 555,
  558, 561, 564, 567, 571, 574, 577, 580,
  583, 586, 589, 593, 596, 599, 602, 606,
  609, 612, 616, 619, 622, 626, 629, 632,
  636, 639, 643, 646, 650, 653, 657, 660,
  664, 668, 671, 675, 679, 682, 686, 690,
  693, 697, 701, 705, 709, 712, 716, 720,
  724, 728, 732, 736, 740, 744, 748, 752,
  756, 760, 764, 769, 773, 777, 781, 785,
```

Syntax

```
  790, 794, 798, 803, 807, 811, 816, 820,
  825, 829, 834, 838, 843, 847, 852, 856,
  861, 866, 870, 875, 880, 885, 890, 894,
  899, 904, 909, 914, 919, 924, 929, 934,
  939, 944, 949, 954, 960, 965, 970, 975,
  981, 986, 991, 997, 1002, 1007, 1013, 1018
};

// Number of fractional bits in the fixed point representation of
// the entries in exponential_table.
const int EXPONENTIAL_TABLE_FRAC_BITS = 9;

// FixedPoint approximation of exp2(fp) = 2^fp
FixedPoint exp2() const {
  int int_part = IntPart();
  int frac_part = FracPart();

  if (PRECISION >= EXPONENTIAL_TABLE_RESOLUTION) {
    frac_part >>= PRECISION - EXPONENTIAL_TABLE_RESOLUTION;
  }
  else {
    frac_part <<= EXPONENTIAL_TABLE_RESOLUTION - PRECISION;
  }

  int frac_exp = (int)(EXPONENTIAL_TABLE[frac_part]);

  if (int_part >= EXPONENTIAL_TABLE_FRAC_BITS - PRECISION) {
    frac_exp <<= int_part + PRECISION - EXPONENTIAL_TABLE_FRAC_BITS;
  }
  else {
    frac_exp >>= EXPONENTIAL_TABLE_FRAC_BITS - PRECISION - int_part;
  }

  FixedPoint res;
  res._data = frac_exp;
  return res;
}

// static functions

// intpart
static int intpart(FixedPoint fp) {
  return fp.IntPart();
}

// fracpart
static int fracpart(FixedPoint fp) {
  return fp.FracPart();
}

// round
static int round(FixedPoint fp) {
  return fp.round();
}

// Right shift
static FixedPoint operator>>(FixedPoint x, unsigned int i) {
  FixedPoint z(x);
  return z >>= i;
}

// Left shift
static FixedPoint operator<<(FixedPoint x, unsigned int i) {
  FixedPoint z(x);
  return z <<= i;
}

// Negation
static FixedPoint operator-() {
  FixedPoint fp(*this);
  fp._data = -fp._data;
  return fp;
}

// Addition
static FixedPoint operator+(FixedPoint x, FixedPoint y) {
  FixedPoint z(x);
```

Syntax

```
    return z += y;
  }

  // Addition of fp and int (right)
  static FixedPoint operator+(FixedPoint x, int y) {
    FixedPoint z(x);
    return z += y;
  }

  // Addition of fp and int (left)
  static FixedPoint operator+(int y, FixedPoint x) {
    FixedPoint z(x);
    return z += y;
  }

  // Subtraction
  static FixedPoint operator-(FixedPoint x, FixedPoint y) {
    FixedPoint z(x);
    return z -= y;
  }

  // Subtraction of int from fp
  static FixedPoint operator-(FixedPoint x, int y) {
    FixedPoint z(x);
    return z -= y;
  }

  // Multiplication
  static FixedPoint operator*(FixedPoint x, FixedPoint y) {
    FixedPoint z(x);
    return z *= y;
  }

  // Multiplication by int (right)
  static FixedPoint operator*(FixedPoint x, int y) {
    FixedPoint z(x);
    return z *= y;
  }

  // Multiplication by int (left)
  static FixedPoint operator*(int y, FixedPoint x) {
    return x * y;
  }

  // Division
  static FixedPoint operator/(FixedPoint x, FixedPoint y) {
    FixedPoint z(x);
    return z /= y;
  }

  // Division by int
  static FixedPoint operator/(FixedPoint x, int y) {
    FixedPoint z(x);
    return z /= y;
  }

  // \brief Minimum
  static FixedPoint min(FixedPoint x, FixedPoint y) {
    return (x <= y ) ? x : y;
  }

  // Maximum
  static FixedPoint max(FixedPoint x, FixedPoint y) {
    return (x >= y ) ? x : y;
  }

  // FixedPoint approximation of exp2(fp) = 2^fp
  static FixedPoint exp2(FixedPoint fp) {
    return fp.exp2();
  }

private:
  int _data;
};
```

8.4.2 Fixed Point Instances

Table 8-25 defines some fixed point types used in the ACE decoder sample code.

Table 8-25: Fixed Point Instances

Syntax

`typedef FixedPoint<10> LNFP;  // lognorm`
`typedef FixedPoint<10> BAFP;  // bit allocation`
`typedef FixedPoint<4> LBFP;  // band size`
`typedef FixedPoint<5> RBFP;  // running balance`
`typedef FixedPoint<11> BQFP;  // Band Quantizer`

9 ACE Decoder

9.1 Overview

An ACE elementary bitstream consists of a sequence of frames, where each frame encodes one or more waveforms for a given specified window in time. In general, ACE frames have temporal dependencies and can only be successfully decoded when a previous frame has been successfully decoded. An elementary bitstream may be partitioned in Group of Frames (GoF), where each GoF is a contiguous set of frames that may be decoded independently of any frames preceding or following the GoF (see Figure 9-1). The first frame in a GoF is referred is referred to as a SYNC frame. The first frame in an ACE elementary bitstream is always a SYNC frame.

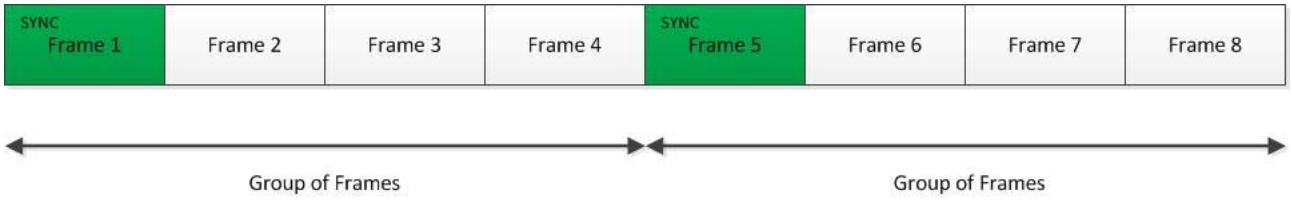

Figure 9-1: ACE Elementary Stream

9.2 Bitstream

9.2.1 Bitstream Data and Method

An ACE bitstream consists of one or more Frames that are read out sequentially, as shown in Table 9-1.

Table 9-1: BitStream Data and Methods

Syntax	Reference

`// A bitstream consists of one or more Frames that are read out sequentially.`
 	
`struct BitStream {`
 	
`  Frame frame[];`
`  void read_bitstream();`
`}`
	9.2.2

9.2.2 read_bitstream

read_bitstream() fetches and process the ACE audio frames in presentation order, as shown in Table 9-2.

Table 9-2: read_bitstream

Syntax	Reference

`read_bitstream() {`
 	
`Frame* previous = NULL;`
 	
`for (int i=0;;i++) {`
`  frame[i].read_frame(previous);`
`  previous = &(frame[i]);`
`  }`
`}`
	9.3.2

9.3 ACE Frames

9.3.1 Frame Data and Methods

A *Frame*, shown in Table 9-3, encapsulates one or more compressed waveforms, logically grouped in *StreamSets*; each *StreamSet* logically groups a number of audible elements (*Streams*) that are intended to be played out together.

Table 9-3: Frame

Syntax	Reference

`struct Frame {`
 	
`  Frame *_previous;`
 	
`  uint _padding;`
`  FrameHeader _header;`
`  StreamSet streamset[];`
 	
`  void read_decode_frame(Frame* previous);`
`}`
	9.3.2

Parameters

- **_previous** - pointer to the previous *Frame*, initialized to NULL at the initialization of a decoder.
- **_padding** - the number of padding bytes. Padding bytes are to be ignored by a compliant decoder.
- **_header** - data that are common for all streamsets in a *Frame*, as well as for streamset-specific data for all streamsets present in a *Frame*.
- **streamset[]** - the actual streamsets within a *Frame*.
- **read_decode_frame()** -read and decode a single *Frame* from a bitstream.

9.3.2 read_decode_frame

The function *read_decode_frame*, shown in Table 9-4, reads and decodes a single frame from a bitstream. This function takes a pointer to the previous *Frame* as input parameter.

Table 9-4: read_decode_frame

Syntax	Reference

`// Read a single Frame from the bitstream`
`//`
`// parameter[in] frame: pointer to the previous frame.`
 	
`void read_decode_frame(Frame* frame) {`
 	
`  // Remember previous frame`
`  _previous = frame;`
 	
`  // read frame header`	

Syntax	Reference

_header.read_frame_header(this);
// read padding info
read_padding()
 	
// read streamsets
for (int i=0;i<_header.num_stream_sets;i++) {
  streamset[i].read_decode_streamset(this,&(_header.streamset_header[i]));
}
 	
// skip padding bytes.
skip(_padding);
}
	9.3.3.1
	
	9.6.2
	
	8.2.2

9.3.3 Function Supporting read_decode_frame

9.3.3.1 read_padding

The function *read_padding*, shown in Table 9-5, reads the number of bytes to be skipped (*_padding*) after completion all of the streamsets in a *Frame*.

Table 9-5: read_padding

Syntax	Reference

// read number of padding bytes from a bitstream
 	
void read_padding() {
 	
_padding = 0;
Table PRIMARY_TABLE = {7,31};
Table ESCAPE_TABLE = {};
 	
if (_header.has_padding) {
  _padding = ReadVLC(PRIMARY_TABLE, ESCAPE_TABLE);
}
}
	8.2.15

9.4 ACE Frame Header

9.4.1 Frame Header Data and Methods

The structure of a Frame Header is shown in Table 9-6. A *FrameHeader* records *Frame* properties that are persistent for every *StreamSet* and every *Stream* contained within a *Frame*. A *Frame* is encoded as a byte-aligned data packet that may contain zero or more padding bytes. Any decoder conforming to the present document shall ignore padding data (if present). Properties that are not explicitly encoded in the bitstream (i.e. not a sync frame) are generally copied or derived from the corresponding data in the previous frame.

Table 9-6: FrameHeader

Syntax

struct FrameHeader {
 	
Frame* frame;
FrameHeader *previous;
 	
bool is_sync;
uint sampling_rate;
uint frame_duration;
bool enable_deemphasis;
uint num_stream_sets;
bool has_padding;
 	
StreamSetHeader streamset_header[];
void read_frame_header(Frame* frame);
}

Parameters

- **frame -** pointer to the containing *Frame*.
- **previous -** pointer to the *FrameHeader* of the previous *Frame*.
- **is_sync -** indicating the start of a Group of Frames.
- **sampling_rate -** the sampling rate of the waveforms encoded in a Frame, with allowed values of 44 100 Hz and 48 000 Hz. The *sampling_rate* value is copied from the previous frame in a non-sync frame. The *sampling_rate* shall be constant for a bitstream.
- **frame_duration -** the duration of the encoded measured in number of samples. The value of *frame_duration* shall be equal to 1 024.
- **enable_deemphasis** - indicating whether de-emphasis processing is enabled for the frame or not.
- **num_stream_sets -** the number of streamsets in a *Frame*. The value of this property is not required to be constant throughout a Group of Frames.
- **has_padding -** whether a *Frame* includes padding data (*has_padding* = ***true***) or not (*has_padding* = ***false***).
- **streamset_header[] -** an array of *num_streamset_headers* streamset headers. The number of streamsets within a Group of Frames need not be constant.
- **read_frame_header() -** read *FrameHeader* from bitstream.

9.4.2 read_frame_header

The function *read_frame_header*, shown in, reads basic information, shared across all streamsets, from a bitstream.

Table 9-7: read_frame_header

Syntax	Reference
`// read the FrameHeader from a bitstream`	
`//`	
`// parameter[in] frame: pointer to the frameheader in the previous frame`	
`void read_frame_header(Frame* containing_frame) {`	
`  align();`	8.2.1
`  // pointer to corresponding Frame`	
`  frame = containing_frame;`	
`  // pointer to previous frame header`	
`  previous = NULL;`	
`  if (frame->_previous) {`	
`    previous = &(frame->_previous._header);`	
`  }`	
`  // read is_sync`	9.4.3.1
`  read_is_sync();`	
`  // Set default values`	
`  if (!is_sync) {  // copy from previous frame`	
`    sampling_rate = previous->sampling_rate;`	
`    frame_duration = previous->frame_duration;`	
`    enable_deemphasis = previous->enable_deemphasis;`	
`  }`	
`  // read from bitstream`	
`  else {`	
`    read_enable_deemphasis();`	9.4.3.2
`    read_frame_duration();`	9.4.3.3
`    read_sampling_rate();`	9.4.3.4
`  }`	
`  read_num_stream_sets();`	9.4.3.5
`  read_has_padding();`	9.4.3.6
`  for (int i=0;i<num_stream_sets;i++) {`	

Syntax	Reference

`    streamset_header[i].read_streamset_header(frame, previous,i);`
`  }`
`}`
	9.5.2

9.4.3 Functions used by read_frame_header

9.4.3.1 read_is_sync

The function *read_is_sync*, shown in Table 9-8, reads the *sync* flag from the bitstream. A *Frame* with the *is_sync* flag set to *true* is referred to as a *sync-Frame*, and indicates the start of a GoF. In a *sync-Frame*, bitstream metadata are not predicted or derived from metadata in a previous *Frame*. The *sync* flag, when *true*, is followed in the bitstream by a reserved Boolean, which should be set to *false*.

Table 9-8: read_is_sync

Syntax	Reference

`// Read the sync from a bitstream`
 	
`read_is_sync() {`
`  is_sync = ReadBool();`
`}`
	8.2.7

9.4.3.2 read_enable_deemphasis

The function *read_enable_deemphasis()*, shown in Table 9-9, reads the enable_deemphasis Boolean flag, which enables (enable_deemphasis = true) or disables de-emphasis processing in the decoder.

Table 9-9: read_enable_deemphasis

Syntax	Reference

`// Read the enable_deemphasis flag`
 	
`void read_enable_deemphasis() {`
`  coded_val = 0;`
`  while (coded_val <= 3){`
`    uint b = ReadBit();`
`    if (b == 0)`
`      break;`
`    coded_val++;`
`  }`
`  switch ( coded_val ) {`
`    case 0:`
`      enable_deemphasis = true;`
`      break;`
`    case 1:`
`      enable_deemphasis = false;`
`      break;`
`    default:`
`      abort("reserved value"); // values 2, 3 are reserved for future use`
`  }`
`}`
	8.2.6

9.4.3.3 read_frame_duration

The function *read_frame_duration()*, shown in Table 9-10, reads the duration of the encoded measured in number of samples. For this version of the decoder the only allowed value is 1 024.

Table 9-10: read_frame_duration

Syntax	Reference

// Read frame_duration from bitstream
 	
void read_frame_duration() {
  uint frame_duration_index = ReadUnary(2);
  if (frame_duration_index == 0) {
  frame_duration = 1024;
  }
  else {
  abort ("reserved value");
  }
}
	8.2.10

9.4.3.4 read_sampling_rate

The function *read_sampling_rate()*, shown in Table 9-11, reads the sampling rate of the waveforms encoded in a Frame, with allowed values of 44 100 Hz and 48 000 Hz. The sampling_rate value is copied from the previous frame in a non-sync frame. The sampling rate is constant for the duration of an ACE bitstream.

Table 9-11: read_sampling_rate

Syntax	Reference

// Read sampling rate from bitstream
 	
void read_sampling_rate() {
 	
  uint sampling_rate_index = ReadUnary(2);
  if (sampling_rate_index == 0) {
  sampling_rate = 48000;
  }
  else if {
  sampling_rate = 44100;
  else {
  abort("reserved value");
}
	8.2.10

9.4.3.5 read_num_stream_sets

The function *read_num_stream_sets*, shown in Table 9-12, reads the number of streamsets within a *Frame*. This value is dynamic and may change from frame to frame.

Table 9-12: read_num_stream_sets

Syntax	Reference

// Read num_stream_sets from bitstream
 	
read_num_stream_sets () {
  num_stream_sets = ReadLimitsVLC() + 1;
}
	8.2.16

9.4.3.6 read_has_padding

The function *read_has_padding*, shown in Table 9-13, reads the *has_padding* flag from the bitstream. If this flag is set, the number of padding bytes will be read from the bitstream before frame parsing continues.

Table 9-13: read_has_padding

Syntax	Reference

// Read padding flag
 	
read_has_padding () {
  has_padding = ReadBool();
}
	8.2.7

9.5 ACE StreamSetHeader

9.5.1 StreamSetHeader Data and Methods

A *StreamSetHeader*, shown in Table 9-14, carries the metadata that are common for all streams included in a *StreamSet*.

Table 9-14: StreamSetHeader

Syntax

```
struct StreamSetHeader {

  Frame* frame;                       // pointer to containing frame
  FrameHeader* frame_header;          // pointer to containing frame header
  StreamSetHeader *previous;          // pointer streamsetheader in previous frame
  uint streamset_index;               // index of streamset in streamset array

  uint streamset_id;                  // unique identifier for streamset
  bool is_predictive;                 // is predictive metadata being used?

  uint num_lfe_stream;                // number of LFE streams
  uint num_mono_streams;              // number of mono streams
  uint num_stereo_streams;            // number of stereo streams
  uint num_lfe_channels[];            // number of LFE channels for each LFE stream

  uint mono_bandwidth_mode[];         // bandwidth mode for mono streams
  uint stereo_bandwidth_mode[];       // bandwidth mode for stereo streams

  uint lfe_stream_payload_size[];     // payload size for LFE streams in bytes
  uint mono_stream_payload_size[];    // payload size for mono streams in bytes
  uint stereo_stream_payload_size[]; // payload size for stereo streams in bytes

  void read_streamset_header(Frame* frame,
                             FrameHeader *containing_frame_header
                             uint index);
}
```

Parameters

- **frame -** pointer to containing *Frame*.
- **frame_header -** pointer to containing *FrameHeader*.
- **previous** - pointer to the corresponding *StreamsetHeader* in the previous *Frame*.
- **streamset_index -** index of streamset in streamset array.
- **streamset_id -** streamset identifier, binding streamsets across frames.
- **is_predictive -** boolean value equal to *true* if streamset metadata is predicted or derived from previous frames.
- **num_lfe_streams -** the number of LFE streams in the streamset.
- **num_mono_streams -** the number of mono streams in the streamset.
- **num_stereo_streams -** the number of stereo streams in the streamset.
- **num_lfe_channels[] -** the number of LFE channels for each of the LFE streams.
- **mono_bandwidth_mode[] -** the bandwidth mode for each of the mono streams.
- **stereo_bandwidth_mode[] -** the bandwidth mode for each of stereo streams.
- **lfe_stream_payload_size[] -** the payload size for each of the LFE streams (bytes).
- **mono_stream_payload_size[] -** the payload size for each of the mono streams (bytes).
- **stereo_stream_payload_size[] -** the payload size for each of the stereo streams (bytes).

- **read_streamset_header() -** read StreamSetHeader from bitstream.

9.5.2 read_streamset_header

The function *read_streamset_header*, shown in Table 9-15, reads streamset header data from a bitstream.

Table 9-15: read_streamset_header

Syntax	Reference

`// read a streamset header from a bitstream`
`//`
`// parameter[in] frame: pointer to containing frame`
`// parameter[in] header: pointer to containing frame header`
`// parameter[in] index: index in the streamset array`
 	
`void read_streamset_header(Frame *containing_frame,`
`FrameHeader* containing_frame_header`
`uint index) {`
 	
`frame = containing_frame;`
`frame_header = containing_frame_header;`
`streamset_index = index;`
 	
`bool is_sync = frame->_header.is_sync;`
 	
`// Read streamset_id`
`read_streamset_id();`
 	
`// Default values`
`previous = NULL;`
`is_predictive = false;`
`if (!is_sync) {`
`uint prev_num_stream_sets = frame_header->previous->num_stream_sets;`
`StreamSetHeader prev_streamset_header[] =`
`frame_header->previous->streamset_header;`
`for (int index = 0; index<previous_num_stream_sets;index++) {`
 	
`if (prev_streamset_header[index].stream_id == stream_id) {`
`// has this streamset been precessed before`
`previous = &previous_streamset_header[index];`
`is_predictive = true;`
`break;`
`}`
`}`
`}`
 	
`if (is_predictive) {`
 	
`num_lfe_stream = previous->num_lfe_streams;`
`num_mono_streama = previous->num_mono_streams;`
`num_stereo_stream = previous-> num_stereo_streams;`
`num_lfe_channels = previous->num_lfe_channels;`
 	
`mono_bandwidth_mode = previous->mono_bandwidth_mode;`
`stereo_bandwidth_mode = previous->stereo_bandwidth_mode;`
`}`
`else {`
`read_stream_information();`
`read_bandwidth_mode();`
`}`
 	
`for( uint i = 0; i < num_lfe_streams; ++i ) {`
`read_lfe_stream_payload_size(i);`
`}`
`for( uint i = 0; i < num_mono_streams; ++i ) {`
`read_mono_stream_payload_size(i);`
`}`
`for( uint i = 0; i < num_stereo_streams; ++i ) {`
`read_stereo_stream_payload_size(i);`
`}`
`}`
	9.5.3.1
	
	9.5.3.2
	9.5.3.6
	
	9.5.3.3
	
	9.5.3.4
	
	9.5.3.5

9.5.3 Functions called from read_streamset_header

9.5.3.1 read_streamset_id

The function *read_streamset_id*, shown in Table 9-16 reads the *streamset_id* from the bitstream. The *streamset_id* binds streamsets across time.

Table 9-16: read_streamset_id

Syntax	Reference

`// read streamset_id`
 	
`void read_streamset_id() {`
`  streamset_id = ReadLimitsVLC();`
`}`
	8.2.16

9.5.3.2 read_stream_information

The function *read_stream_information*, shown in Table 9-17, reads the composition of a streamset as a set of *LFEStreams*, *MonoStreams*, and *StereoStreams* from the bitstream.

Table 9-17: read_stream_information

Syntax	Reference

`// Read stream information from the bitstream`
`//`
`// 1. The first 10 composition code values indicate common compositions.`
`// 2. The 11th composition code value indicates an explicit composition.`
`// 3. The remaining composition code values are reserved.`
 	
`void read_stream_information() {`
 	
`  uint stream_info_code = ReadLimitsVLC();`
 	
`  switch( stream_info_code ) {`
 	
`    case 0: // 0b0 -> 1p0`
`      num_lfe_streams = 0;`
`      num_mono_streams = 1;   // mono streams`
`      num_stereo_streams = 0; // stereo streams`
`      break;`
 	
`    case 1: // 0b10 -> 2p0`
`      num_lfe_streams = 0;`
`      num_mono_streams = 0;   // mono streams`
`      num_stereo_streams = 1; // stereo streams`
`      break;`
 	
`    case 2: // 0b1100 -> 5p1`
`      num_lfe_streams = 1;`
`      num_mono_streams = 1;   // mono streams`
`      num_stereo_streams = 2; // stereo streams`
`      break;`
 	
`    case 3: // 0b1101 -> 7p1`
`      num_lfe_streams = 1;`
`      num_mono_streams = 1;   // mono streams`
`      num_stereo_streams = 3; // stereo streams`
`      break;`
 	
`    case 4: // 0b1110 -> 11p1`
`      num_lfe_streams = 1;`
`      num_mono_streams = 1;   // mono streams`
`      num_stereo_streams = 5; // stereo streams`
`      break;`
 	
`    case 5: // 0b111100 -> 5p0`
`      num_lfe_streams = 0;`
`      num_mono_streams = 1;   // mono streams`
`      num_stereo_streams = 2; // stereo streams`
`      break;`
	8.2.16

Syntax	Reference

`    case 6: // 0b111101 -> 8p0`
`      num_lfe_streams = 0;`
`      num_mono_streams = 0;   // mono streams`
`      num_stereo_streams = 4; // stereo streams`
`      break;`	

`    case 7: // 0b111110 -> 9p1`
`      num_lfe_streams = 1;`
`      num_mono_streams = 1;   // mono streams`
`      num_stereo_streams = 4; // stereo streams`
`      break;`	

`    case 8: // 0b11111100 -> 8p1`
`      num_lfe_streams = 1;`
`      num_mono_streams = 2;   // mono streams`
`      num_stereo_streams = 3; // stereo streams`
`      break;`	

`    case 9: // 0b11111101 -> 10p1`
`      num_lfe_streams = 1;`
`      num_mono_streams = 2;   // mono streams`
`      num_stereo_streams = 4; // stereo streams`
`      break;`	

`    case 10 // (code=0b11111110) explicit listing`	

`      num_lfe_streams = ReadLimitsVLC();`
`      for (int i = 0; i < num_lfe_streams; i++) {`
`        num_lfe_channels[i] = 1 + ReadLimitsVLC();`
`      }`
	8.2.16

`      num_mono_streams = ReadLimitsVLC();`	8.2.16
`      num_stereo_streams = ReadLimitsVLC();` `      break;`	8.2.16

`    default:`
`      abort("reserved value");`
`  }`
`}`
	8.2.11

9.5.3.3 read_lfe_stream_payload_size

The function *read_lfe_payload_size*, shown in Table 9-18, reads an LFE payload size from the bitstream.

Table 9-18: read_lfe_stream_payload_size

Syntax	Reference

`// read an LFE payload size from a bitstream`
 	
`void read_lfe_stream_payload_size(uint stream_idx) {`
 	
`  lfe_stream_payload_size[stream_idx] = ReadNonUniformFiveTen();`
`}`
	8.2.17

9.5.3.4 read_mono_stream_payload_size

The function *read_mono_stream_payload_size*, shown in Table 9-19, reads a *MonoStream* payload size from the bitstream.

Table 9-19: read_mono_stream_payload_size

Syntax	Reference

`// read a MonoStream payload size from a bitstream`
 	
`void read_mono_stream_payload_size(uint stream_idx) {`
`  Table PRIMARY_TABLE[] = {2,7,9,13};`
`  Table ESCAPE_TABLE[] = {24,32};`
`  mono_stream_payload_size[stream_idx] = ReadVLC(PRIMARY_TABLE,ESCAPE_TABLE);`
`}`
	8.2.15

9.5.3.5 read_stereo_stream_payload_size

The function *read_stereo_stream_payload_size*, shown in Table 9-20, reads a *StereoStream* payload size from the bitstream.

Table 9-20: read_stereo_stream_payload_size

Syntax	Reference

`// read an StereoStream payload size from a bitstream`
 	
`void read_stereo_stream_payload_size(uint stream_idx) {`
`  Table PRIMARY_TABLE[] = {2,7,9,13};`
`  Table ESCAPE_TABLE[] = {24,32};`
`  stereo_stream_payload_size[stream_idx] = ReadVLC(PRIMARY_TABLE,ESCAPE_TABLE);`
`}`
	8.2.15

9.5.3.6 read_stream_bandwidth_mode

The function *read_stream_bandwidth_mode*, shown in Table 9-21, reads bandwidth mode information from a bitstream. The bandwidth mode and determines the number of bands that are coded (*num_effective_bands*). *bandwidth_mode* does not apply to LFE streams.

Table 9-21: read_stream_bandwidth_mode

Syntax	Reference

`// read bandwidth modes from a bitstream`
 	
`void read_stream_bandwidth_mode() {`
 	
`  if( (num_mono_streams + num_stereo_streams ) == 1 ) {  // single bandwidth mode`
`  uint bandwidth_mode = ReadUint(2);`
`  mono_bandwidth_mode[0] = bandwidth_mode;`
`  stereo_bandwidth_mode[0] = bandwidth_mode;`
`  }`
`  else {  // separate mono and stereo bandwidth mode`
`  bool bandwidth_signaling_mode = ReadBool();`
`  if ( bandwidth_signaling_mode == 0 )  // per stream-type bandwidth mode`
`  {`
`  uint mono_bandwidth_mode = ReadUint(2);`
`  for( uint i = 0; i < num_mono_streams; ++i ) {`
`  mono_bandwidth_mode[i] = mono_bandwidth_mode;`
`  }`
`  uint stereo bandwidth_mode = ReadUint(2);`
`  for( uint i = 0; i < num_stereo_streams; ++i ) {`
`  stereo_bandwidth_mode[i] = stereo_bandwidth_mode;`
`  }`
`  }`
`  else  // per stream bandwidth mode`
`  {`
`  for( uint i = 0; i < num_mono_streams; ++i ) {`
`  mono_bandwidth_mode[i] = ReadUint(2);`
`  }`
`  for( uint i = 0; i < num_stereo_streams; ++i ) {`
`  stereo_bandwidth_mode[i] = ReadUint(2);`
`  }`
`  }`
`  }`
`}`
	8.2.8
	8.2.7
	8.2.8
	8.2.8
	8.2.8
	8.2.8

9.6 ACE Stream Set

9.6.1 Stream Set Data and Methods

A *StreamSet*, shown in Table 9-22, aggregates one or more waveforms that are intended to be played back together. The composition of a *StreamSet* is recorded in its associated header.

Table 9-22: AceStreamSet

Syntax

`struct AceStreamSet {`
 	
`  Frame* _frame;`
`  StreamSetHeader* _header;`
`  StreamSet* _previous;`
 	
`  LFEStream _lfe_streams[];`
`  MonoStream _mono_streams[];`
`  StereoStream _stereo_streams[];`
 	
`  float _lfe_buffers[][];  // buffers carrying LFE samples`
`  float _mono_buffers[][];  // buffers encoded as mono`
`  float _stereo_buffers[][];  // buffers encoded as stereo`
 	
`  void read_decode_streamset(Frame* frame, StreamSetHeader, uint index);`
`}`

Parameters

_frame - pointer to the containing frame.

_header - pointer to the streamset header for this streamset.

_previous - pointer to the streamset in the previous frame (same stream_id).

_lfe_streams - LFE streams in the streamset.

_mono_streams - mono streams in the streamset.

_stereo_streams - stereo streams in the streamset.

_lfe_buffers - the decoded waveforms for LFE streams, with one or more channels per LFE stream.

_mono_buffers - the decoded waveforms for mono streams, with one channel per mono stream.

_stereo_buffers - the decoded waveforms for mono streams, with two channels per stereo stream.

read_streamset() - read streamset from bitstream.

9.6.2 read_decode_streamset

The function *read_decode streamset*, shown in Table 9-23, reads and decodes a *StreamSet* from a bitstream by reading and decoding all of its included streams (LFE, mono, stereo).

Table 9-23: read_decode_streamset

Syntax	Reference

`// reading a streamset from a bitstream`
 	
`void read_decode_streamset(Frame* frame, StreamSetHeader header, uint index) {`
 	
`  _frame = frame;`
`  _header = header;`
 	
`  // pointer to streamset in previous frame`
`  _previous = NULL;`
`  if (_header->previous) {`
`    uint prev_streamset_index = _header->previous->streamset_index;`
`    _previous = (frame->_previous).streamset[prev_streamset_index];`
`  }`
 	
`  uint num_lfe_stream = _header->num_lfe_streams;`
`  uint num_mono_streams = _header->num_mono_streams;`
`  uint num_stereo_streams = _header->num_stereo_streams;`
 	
`  // assign LFE buffers and process`
`  float lfe_buffers[][] = _lfe_buffers;`
`  for (int i=0;i<num_lfe_streams;i++) {`	

Syntax	Reference
`    lfe_stream[i].read_decode_lfe_stream(frame,this,lfe_buffers,i);`	9.7.2
`    lfe_buffers += _header.num_lfe_channels[i];`	
`  }`	
`  // assign mono channels and process`	
`  float mono_buffers[][] = _mono_ buffers;`	
`  for (int i=0;i<num_decode_mono_streams;i++) {`	
`    mono_stream[i].read_decode_mono_stream(frame,this,mono_buffers,i);`	9.8.2
`    mono_buffers += 1;`	
`  }`	
`  // assign stereo channels and process`	
`  float stereo_buffers[][] = _stereo_channel;`	
`  for (int i=0;i<num_stereo_streams;i++) {`	
`    stereo_stream[i].read_decode_stereo_stream(frame,this,stereo_buffers,i);`	9.8.3
`    stereo_buffers += 2;`	
`  }`	
`}`	

9.7 ACE LFEStreams

9.7.1 LFE Stream Data and Methods

An *LFEStream* holds one or more *LFEChannels*. Parsing an *LFEStream* amounts to parsing and decoding its included *LFEChannels*. An *LFEStream* contains the data listed in Table 9-24.

Table 9-24: LFEStream

Syntax
`struct LFEStream {`
`  Frame* _frame;`
`  StreamSet* _streamset;`
`  LFEStream *_previous;`
`  uint _stream_index;`
`  LFEChannel _lfe_channel[];`
`  void read_decode_lfe_stream(Frame*frame,`
`                      StreamSet* streamset,`
`                      uint stream_index,`
`                      float lfe_buffer[][]);`
`}`

Parameters

- **_frame -** pointer to the containing frame.
- **_streamset -** pointer to the containing streamset.
- **_previous -** pointer to the LFE stream in the previous frame.
- **_stream_index -** index of the LFE stream.
- **_lfe_channel[] -** list of channels in the LFE stream.
- **read_decode_lfe_stream() -** read an LFE stream from bitstream.

9.7.2 read_decode_lfe_stream

The function *read_lfe_decode_stream*, shown in Table 9-25, parses and decodes an LFE stream by parsing and decoding its included LFE channels.

Table 9-25: read_decode_lfe_stream

Syntax	Reference

`// Parse an LFE stream from the bitstream`
`//`
`// parameter[in] frame: pointer to the current frame`
`// parameter[in] stream_set: pointer to the containing stream set.`
`// parameter[in] stream_index: index _lfe_stream array in stream set.`
`// parameter[out] lfe_channel: PCM output buffers`
 	
`void read_decode_lfe_stream(Frame* frame,`
`StreamSet* streamset,`
`uint stream_index,`
`float lfe_buffers[]) {`
 	
`_frame = frame;`
`_streamset = streamset;`
`_stream_index = stream_index;`
`_previous = NULL;`
`_lfe_channel = lfe_channel;`
 	
`if (_streamset->_previous) {`
`StreamSet* prev_streamset = _stream_set->_previous;`
`_previous = &(prev_stream_set->_lfe_stream[_stream_index]);`
`}`
 	
`uint num_lfe_channels[] = (_stream_set->_header).num_lfe_channels;`
`for (int i=0;I < num_lfe_channels[_stream_index];i++) {`
 	
`_lfe_channel[i].read_decode_lfe_channel(_frame, this,_lfe_buffers[i],i);`
`}`
`}`
	9.7.3.2

9.7.3 LFEChannel

9.7.3.1 LFEChannel Data and Methods

An *LFEChannel* contains the data listed in Table 9-26.

Table 9-26: LFEChannel

Syntax

`struct LFEChannel {`
 	
`// Reference pointers`
`Frame* _frame; // pointer to current frame`
`LFEStream* _stream; // pointer to stream`
`LFEChannel* _previous; // pointer to LFEChannel in previous frame`
`uint _channel_index; // index in the LFEChannel array`
 	
`// derived properties`
`uint _lfe_mode;`
`float _stepsize;`
`uint _frame_duration;`
`uint _decimated_frame_duration;`
 	
`// Properties from bitstream`
`uint _resolution;`
`uint _savings;`
`uint _dbnorm;`
`float _predictor[2];`
 	
`// sample values`
`float _lfe_decimated_buffer[];`
`float _lfe_buffer[];`
 	
`// main methods`
`void read_decode_lfe_channel(Frame* frame,`
`LFEStream* lfe_stream,`
`uint channel_index,`
`float[] lfe_buffer); // bitstream parsing`

Syntax

`  void decode_lfe_channel();                    // decoding to output`
`}`

Parameters

- **_frame -** pointer to containing frame.
- **_stream** - pointer to *LFEStream.*
- **_previous -** pointer to *LFEChannel* in previous frame.
- **_channel_index** - index of channel in *LFEChannel* array.
- **_lfe_mode -** the operating mode of the LFEStream decoder. When *_is_predictive = false*, the predictive mode *kLFEPredictMode* is prohibited.
- **_resolution -** the default number of quantization levels for the reconstruction of sample values. The allowed values are 8, 10 and 12.
- **_savings -** adjustment of the actual number of quantization levels for the reconstruction of sample values. The allowed values for *_savings* are 0, 2, 3 and 4.
- **_dbnorm -** the energy of the reconstructed signal.
- **_predictor -** the two prediction coefficients for LFE sample reconstruction from residuals.
- **_decimated_frame_duration -** the number of decimated and encoded LFE samples is, and is computed from frame_duration (division by 64).
- **_lfe_decimated_buffer -** the buffer of absolute sample index values *(_lfe_mode* != *kLFEPredictMode*) or residual samples index values (all other modes). Output sample values are computed by inverse quantization and/or prediction. The values in this vector are decimated with respect to output values in *_lfe_buffer*.
- **_lfe_buffer -**the output buffer for the LFE channel, obtained by upsampling of *_lfe_decimated_buffer*.
- **read_lfe_channel() -** read an LFEChannel from the bitstream.
- **decode_lfe_channel() -** decode an LFEChannel.

9.7.3.2 read_decode_lfe_channel

The function *read_decode_lfe_channel*, Table 9-27, reads metadata for a single LFE channel. This function ends with *decode_lfe_channel* reconstructing PCM values from the data that has been read from the bitstream.

Table 9-27: read_decode_lfe_channel

Syntax	Reference

`read_lfe_channel(Frame* frame,`
`                  LFEStream* stream,`
`                  uint channel_index`
`                  float lfe_channel[]){`
 	
`  const uint LFE_DECIMATION_FACTOR = 64;   // decimation factor for the LFE signal`
`  const uint LFE_ABSOLUTE_THRESHOLD = 45;`
`  const uint LFE_REDUCED_THRESHOLD = 60;`
`  const uint LFE_RESOLUTION_DECREMENT = 4;`
`  const float LFE_NORMALIZED_STEPSIZE = 4.48;`
 	
`  // pointer to current frame`
`  _frame = frame;`
`  // pointer to stream`
`  _stream = stream;`
`  // channel index`
`  _channel_index = channel_index;`
`  // PCM buffer`
`  _lfe_buffer = lfe_buffer;`
`  set(_lfe_buffer,0);`	

Syntax	Reference
// pointer to LFE channel in previous frame	
_previous = NULL;	
if (stream->_previous) {	
_previous = &(stream->_previous->_lfe_channel[_channel_index]);	
}	
// decimated frame duration	
_frame_duration = _frame->_header.frame_duration;	
_decimated_frame_duration = frame_duration / LFE_DECIMATION_FACTOR;	
read_resolution()	9.7.3.3.1
read_savings();	9.7.3.3.2
read_dbnorm();	9.7.3.3.3
// predicting from residuals	
if (_savings != 0) {	
read_predictor();	9.7.3.3.4
_resolution -= _savings;	9.7.3.3.1
lfe_decimated_channel();	9.7.3.3.5
_lfe_mode = kLFEPredictMode;	9.7.3.2
}	
// using non-predicted values	
else {	9.7.3.3.5
if (_dbnorm <= LFE_ABSOLUTE_THRESHOLD) {	
read_lfe_decimated_buffer();	
_lfe_mode = kLFEAbsoluteMode;	9.7.3.2
}	
else if (_dbnorm <= LFE_REDUCED_THRESHOLD) {	9.7.3.3.5
_resolution -= LFE_RESOLUTION_DECREMENT;	
read_lfe_decimated_buffer();	
_lfe_mode = kLFEReducedMode;	9.7.3.2
}	
else {	
_lfe_mode = kLFESyntheticMode;	
}	
}	
float factor = db_to_linear(_dbnorm) * ((1 << _resolution) >> 1);	9.11.4.5
_stepsize = LFE_NORMALIZED_STEPSIZE / factor;	
decode_lfe_channel();	9.11.3
}	

9.7.3.3 Functions Called by read_lfe_channel

9.7.3.3.1 read_resolution

The function *read_resolution* reads the quantization resolution from the bitstream.

Table 9-28: read_resolution

Syntax	Reference
// Read quantization resolution from the bitstream	
read_resolution() {	
const uint RESOLUTION_TABLE[] = {8,10,12};	
_resolution = RESOLUTION_TABLE[ReadUniform(3)];	8.2.11
}	

9.7.3.3.2 read_savings

The function *read_savings* reads the adjustment to resolution value for predictive decoding from the bitstream.

Table 9-29: read_savings

Syntax	Reference
// Read adjustment to quantization resolution from the bitstream.	
read_savings() {	
LFE_SAVINGS_BITS = 2;	
_savings = ReadUint(LFE_SAVINGS_BITS);	8.2.8

Syntax	Reference
`  _savings = (_savings == 0) ? 0 : _savings + 1;` `}`	

9.7.3.3.3 read_dbnorm

The function *dbnorm* reads the lognorm of the decimated LFE channel from the bitstream.

Table 9-30: read_dbnorm

Syntax	Reference

`// Read the lognorm of the decimated LFE channel from the bitstream.`
 	
`read_dbnorm() {`
 	
`  uint LFE_NORM_BITS = 6;`
`  _dbnorm = ReadUint(LFE_NORM_BITS);`
`}`
	8.2.8

9.7.3.3.4 read_predictor

The function *read_predictor*() reads the prediction coefficient for predictive decoding from the bitstream.

Table 9-31: read_predictor

Syntax	Reference

`// Read the prediction coefficients from the bitstream`
 	
`read_predictor() {`
 	
`  const uint LFE_PREDICTION_ORDER = 2;`
`  const uint LFE_LAR_INDEX_ALPHABET_SIZE = 255;`
 	
`  for (int i=0;i<LFE_PREDICTION_ORDER;i++) {`
`    int index = (int) ReadUniform(LFE_LAR_INDEX_ALPHABET_SIZE);`
`    if (index &1) {`
`      _predictor[i] = -PREDICTOR_QUANTIZATION_COEFF[(index+1)/2];`
`    }`
`    else {`
`      _predictor[i] = +PREDICTOR_QUANTIZATION_COEFF[(index+0)/2];`
`    }`
`  }`
 	
`  _predictor[0] = -_predictor[0] / (1 + _predictor[1]);`
`  _predictor[1] = -_predictor[1];`
`}`
	8.2.11

9.7.3.3.5 read_lfe_decimated_buffer

The function *read_lfe_decimated_buffer*() reads sample values (in non-predictive mode) or residual values (in predictive mode) from the bitstream.

Table 9-32: read_lfe_decimated_buffer

Syntax	Reference

`// Read samples or residuals from the bitstream`
 	
`read_lfe_decimated_buffer() {`
 	
`  // Read Golomb k-parameter`
`  uint kparam;`
`  uint kparam = ReadUniform(_resolution);`
 	
`  // Golomb decoding and Rice mapping`
`  for (int i=0;i<_decimated_frame_duration;i++) {`
`    uint sample = ReadGolomb(kparam,(1U << _resolution));`
`    _lfe_decimated_buffer[i] = (sample & 1) ? -(sample + 1)/2 : sample/2;`
	8.2.11
	
	8.2.12

Syntax	Reference
} }	

9.8 ACE Streams

9.8.1 Stream Data and Methods

A *Stream* contains the data listed in Table 9-33.

Table 9-33: Stream

Syntax	Reference

struct Stream {
 	
  Frame* _frame;
  StreamSet* _streamset;
  Stream* _previous;
  uint stream_index;
 	
  float _stream_buffer[][];
 	
  uint _total_bits;
 	
  BandsConfig* _config;
	10.2.3.1

  uint _frame_duration;
  bool enable_deemphasis;
 	
  uint _num_stream_channels;
  uint _num_effective_bands;
  uint _num_coded_bands;
 	
  bool _is_predictive;
  bool _is_stereo;
  uint _coding_mode;
  bool _is_short;
  bool _is_highres;
  bool _shf_enabled;
  bool _thf_enabled;
 	
  // effective channels
  uint _first_effective_channel;
  uint _last_effective_channel;
  uint _num_effective_channels;
 	
  // time-frequency structure
  uint _num_band_blocks[];
 	
  // recondtioning
  uint _reconditoning_level;
 	
  // Long term synthesis
  uint _lts_lag;
  float _lts_coefficients[];
 	
  // Lognorm
  LNFP _lognorm[][];
  LNFP _lognorm_with_offset[];
 	
  // Bit Allocation
  uint _alloc[];
  uint _alloc_model;
  uint _alloc_model_parameter;
  uint _alloc_max[];
  uint _alloc_min[];
  uint _alloc_delta[];
  uint _min_guaranteed_alloc;
 	
  // stereo
  uint _num_ms_mono_bands;
  uint _stereo_flag[];	

Syntax	Reference

uint _full_frame_stereo_coding_flag;
uint _lr_alloc_region_mode[];
 	
// lognorm refinement
uint _first_refinement_alloc[];
uint _saved_late_refinement_alloc[];
uint _late_refinement_alloc[];
 	
// deemphasis
float _deemphasis_prev[];
 	
void read_decode_stream(Frame* frame, StreamSet* streamset, uint index, float buffer[][]);
}	

Parameters

- **_frame** - pointer to the containing frame.
- **_streamset** - pointer to the containing streamset.
- **_previous** - pointer to the stream in the previous frame.
- **_stream_index** - index in the array of streams in the streamset.
- **_stream_buffers** - buffers holding the decoded PCM samples.
- **_config** - band configuration, including definition of spectral bands.
- **_total_bits** - the number of bits available for reading and decoding.
- **_frame_duration** - frame duration in number of samples
- **_enable_deemphasis** - Boolean flag indicating whether de-emphasis processing is enabled for this stream.
- **_is_predictive** - is metadata predictively encoded?
- **_is_stereo** - if *true*, this stream contains two channels that are jointly encoded (stereo stream). Otherwise, this stream contains precisely one channel (mono stream).
- **_coding_mode** - signalling which channels have spectral data encoded in the bitstream: none, all, left only, right only.
- **_first_effective_channel** - the (index of the) first channel that is encoded in the bitstream.
- **_last_effective_channel** - the (index of the) last channel that is encoded in the bitstream.
- **_num_stream_channels** - number of channels in the stream: one for mono, two for stereo.
- **_num_effective_channels** - number of effective channels.
- **_num_effective_bands** - number of effective bands, determined by bandwidth.
- **_num_coded_bands** - number of bands encoded in the bitstream.
- **_is_short** - if *true*, a reduced frequency resolution of 128 spectral lines is used, associated with either 4 or 8 MDCT blocks. If *false*, the frequency resolution is equal to the frame duration, with a single MDCT block per frame. The former is referred to as a short frame, whereas the latter is referred to as a long frame.
- **_is_highres** - if *true*, the vector quantizer used to quantize and encode bands' spectral shapes operates in a higher resolution mode. If *false*, the quantizer operates in normal resolution mode.
- **_thf_enabled** - if *true*, temporal hole filling is recommended.
- **_reconditioning_level** - this property holds the smoothing level value, used in the assignment of byte quota to spectral bands.

- **_lts_lag -** the lag for the long term synthesis filter.
- **_lts_coefficients -** the filter tabs for the long term synthesis filter.
- **_lognorm** - log 2 of spectral band norms, one value per channel, per band.
- **_lognorm_with_offset** - log2 of spectral bandnorm offset with mean_lognorm.
- **_alloc** - bit allocation per band.
- **_alloc_model** - bit allocation model.
- **_alloc_max** - maximum allocation per band.
- **_alloc_min** - minimum allocation per band that can be used for VQ index coding.
- **_alloc_delta** - differential allocation values.
- **_min_guaranteed_alloc** - bit allocation guaranteed to each effective band.
- **_alloc_model_parameter** - bit allocation model parameter.
- **_unallocated_bits** - number of bits not allocated by allocation module.
- **_num_ms_mono_bands** - number of bands that are midside-mono coded.
- **_stereo_flag** - per band stereo flag.
- **_full_frame_stereo_coding_flag** - flagging a single stereo coding decision.
- **_lr_alloc_region_mode** - stereo allocation mode per region.
- **_first_refinement_alloc -** the number of refinement bits for first lognorm refinement, one value per band.
- **_saved_late_refinement_alloc** - the number of bits saved for late lognorm refinement during band reconstruction, one value per band.
- **_late refinement_alloc** - the final number of refinement bits for late lognorm refinement, one value per band, derived from *_saved_late_refinement_alloc* and from the number of unused bits remaining after band reconstruction.

9.8.2 read_decode_mono_stream

A *MonoStream* is specific instance of general Stream, with *_is_stereo* set to *false*, encoding a single waveform, as shown in Table 9-34.

Table 9-34: MonoStream

Syntax
`struct MonoStream : Stream { _is_stereo = false; }`

The call to *read_ stream()* is shown in Table 9-35.

Table 9-35: read_decode_mono_stream

Syntax	Reference

`// Parse a mono stream from the bitstream`
`//`
`// parameter[in] frame: pointer to current frame`
`// parameter[in] streamset: pointer to containing stream_set`
`// parameter[in] stream_index: stream index in containing stream_set`
`// parameter[out] buffer: pointer to buffer array (single buffer only)`
 	
`void read_decode_mono_stream(Frame* frame,`
`StreamSet* stream_set,`
`uint stream_index,`	

float buffer[][]) {
  read_decode_stream(frame, streamset, stream_index, false, buffer);
}
	9.8.4

9.8.3 read_decode_stereo_stream

A *StereoStream* is specific instance of general Stream, with *_is_stereo* set to *true*, encoding exactly two waveforms, as shown in Table 9-36.

Table 9-36: StereoStream

Syntax
struct StereoStream : Stream { _is_stereo = true; }

The call to *read_decode_stream()* is shown in Table 9-37.

Table 9-37: read_decode_stereo_stream

Syntax	Reference

// Parse a stereo stream from the bitstream
//
// parameter[in] frame: pointer to current frame
// parameter[in] streamset: pointer to containing stream_set
// parameter[in] stream_index: stream index in containing stream_set
// parameter[out] buffer: pointer to buffer array (two buffer)
 	
void read_decode_stereo_stream(Frame* frame,
StreamSet* stream_set,
uint stream_index,
float buffer[][]) {
  read_decode_stream(frame, streamset, stream_index, true, buffer);
}
	9.8.4

9.8.4 read_decode_stream

The function *read_decode_stream()*, in Table 9-38, reads and decodes a single stream from a bitstream.

Table 9-38: read_decode_stream

Syntax	Reference

// Parse a stream from the bitstream
//
// parameter[in] frame: pointer to current frame
// parameter[in] stream_set: pointer to containing stream_set
// parameter[in] stream_index: stream index in containing stream_set
// parameter[in] is_stereo: is this a stereo stream?
// parameter[out] buffers: pointer to buffer array
 	
read_decode_stream(Frame* frame,
StreamSet* streamset,
uint stream_index,
bool is_stereo,
float buffers[][]) {
  const uint ALLOC_MODEL_PARAMETER_DEFAULT = 10;
  // Reset bitstream bit counter
  BitsUsed(0);
  // Reset bitstream capacity
  if (_is_stereo) {
  _total_bits = (stream_set->_header).stereo_stream_payload_size[_stream_index] << 3;
  }
  else {
  _total_bits = (stream_set->_header).mono_stream_payload_size[_stream_index] << 3;
  }
  BitsTotal(_total_bits);
  // Is this a stereo stream?
  _is_stereo = is_stereo;
  // set number of stream channels
  if (is_stereo) {
	8.2.3
	
	8.2.5

Syntax	Reference

`  _num_stream_channels = 2;`
`  }`
`  else {`
`  _num_stream_channels = 1;`
`  }`
`  // assign the output PCM channels and initialize`
`  for (int i=0;i<_num_stream_channels) {`
`  _stream_buffers[i] = buffers[i];`
`  set(_stream_buffer[i],0);`
`  set(_lognorm_with_offset[i], LOGNORM_MINUS_INFINITY);`
`  set(_lognorm[i], LOGNORM_MINUS_INFINITY);`
`  }`
`  // pointer to frame`
`  _frame = frame;`
`  // frame duration`
`  _frame_duration = (_frame->_header).frame_duration;`
`  // enable deemphasis flag`
`  _enable_deemphasis = (frame->_header).enable_deemphasis;`
`  // pointer to stream_set`
`  _streamset = streamset;`
`  // index to stream`
`  _stream_index = stream_index;`
`  // is_predictive`
`  _is_predictive = (_streamset -> _header).is_predictive;`
`  // pointer to stream in previous frame`
`  _previous = NULL;`
`  if (_streamset->_previous) {`
`  if (_is_stereo) {`
`  _previous = &((_streamset->_previous)._stereo_stream[_stream_index]);`
`  else {`
`  _previous = &((_streamset->_previous)._mono_stream[_stream_index]);`
`  }`
`  }`
`  // Band configuration`
`  _config = NULL;`
`  if ((_frame->_header).sampling_rate = 44100) {`
`  _config = &(BANDS_CONFIG_44p1KHZ);`
`  }`
`  if ((_frame->_header).samplaing_rate = 48000) {`
`  _config = &(BANDS_CONFIG_48KHZ);`
`  }`
`  // number of effective bands`
`  const uint bandwidth_mode_table[] = {17, 19, 21, 22};`
`  if (_is_stereo) {`
`  _num_effective_bands = bandwidth_mode_table[stereo_bandwidth_mode[_stream_index]];`
`  }`
`  else {`
`  _num_effective_bands = bandwidth_mode_table[mono_bandwidth_mode[_stream_index]];`
`  }`
 	
`  _num_coded_bands = _num_effective_bands;`
`  _alloc_model_parameter = ALLOC_MODEL_PARAMETER_DEFAULT;`
`  if (_previous & !_is_predictive) {`
`  _num_coded_bands = _previous -> _num_coded_bands);`
`  }`
`  set(_deemphasis_prev,0);`
`  read_coding_mode();`
`// compute effective channels`
`  compute_effective_channels();`
`// each effective band is guaranteed a minimum bit allocation (if this is all it gets,`
`// it will be used for lognorm refinement).`
`  _min_guaranteed_alloc = _num_effective_channels;`
`  // set allocation boundaries, depdendent on`
`  // number of effective channels`
`  for (uint band = 0; band < NUM_ACE_BANDS; band++) {`
`  _alloc_max[band] = _config -> alloc_max[band] * _num_effective_channels;`
`  if (_is_stereo) {`
`  _alloc_max[band] += _config -> bandsize[band];`
`  }`
`  _alloc_min[band] = max(_num_stream_channels, _config -> alloc_min[band]);`
`  }`
`  // read long term synthesis`
`  read_long_term_synthesis();`
`  if (_coding_mode == kCodingOff) {`
`  // stream is not coded`
`  _alloc_model_parameter = ALLOC_MODEL_PARAMETER_DEFAULT;`
`  }`
	10.2.3.2.2
	10.2.3.2.1
	9.9.1
	9.9.2
	9.9.3
	10.2.2

Syntax	Reference

else {
// process if stream is coded
// vector quantizer resolution mode
read_vq_resolution_mode();
	9.9.4

// MDCT mode
read_mdct_mode();
	9.9.5

//read SHF recommendation
read_shf_enabled();
	9.9.6

// read THF recommendation
if (_is_short) {
read_thf_enabled();
	9.9.7

}
// Initial primary lognorm values
read_lognorm_primary();
	9.9.8

read_num_band_blocks();	9.9.9
read_reconditioning_level();	9.9.11
read_allocation();	9.9.12
do_first_lognorm_refinement();	9.9.14

// Done with parsing. Decoding starts here.
// Reconstruct normalized spectral bands
read_all_bands();
	9.10.2.2

// Any remaining bits are assigned to lognorm refinement
do_late_lognorm_refinement();
	9.9.16

// apply lognorm
apply_lognorms();
	9.9.18

// Optionally, apply temporal hole fill
do_temporal_hole_fill();
	9.10.7.3

// apply inverse IMDCT
// note that IMDCT application reconstructs the
// time-domain representation of the previous frame.
// any additional processing applies to the previous
// previous frame.
uint length = _frame_duration; // buffer length
uint N_curr = _frame_duration; // transform size for current buffer
if (_is_short) {
N_curr = _frame_duration >>= 3;
}
uint N_prev = _frame_duration; ; // transform size for previous buffer
if (_previous->_is_short) {
N_prev >>= 3;
}
float curr_vector = _buffers; // buffer in current frame
float prev_vector = _previous -> _buffers; // buffer in previous frame
do_windowed_imdct(length,N_prev,N_curr,prev_vector, curr_vector);
	9.10.5.2

}
// apply long term synthesis to previous frame
// (current frame still waiting for time-domain completion)
_previous-> do_longterm_synthesis(_previous_buffers,_buffers);
	9.10.6.3

// apply de-emphasis processing (if enabled) to previous frame
// (current frame still waiting for time-domain completion)
if(previous->enable_deemphasis) {
_previous -> do_deemphasis();
	9.10.8.2

} return; }	

9.9 Stream Preparation

9.9.1 read_coding_mode

The function *read_coding_mode()*, shown in Table 9-39, reads the coding mode from the bitstream. When *_coding_mode* is equal to *kCodingOff* no spectral coefficients are encoded in the bitstream.

Table 9-39: read_coding_mode

Syntax	Reference

`// read coding mode`
 	
`uint read_coding_mode() {`
`  const uint CODING_MODE_ALPHA = 4;`
`  return ReadUnary(CODING_MODE_ALPHA);`
`}`
	8.2.10

9.9.2 compute_effective_channels

The function *compute_effective_channels()*, shown in Table 9-40, determines the channels that are effectively encoded in the bitstream.

Table 9-40: compute_effective_channels

Syntax

`// Determine the first and last channel that are effectively coded in the bit stream,`
`// depending on the coding mode and the overall number of stream channels.`
`//`
 	
`void compute_effective_channels() {`
 	
`  if ( !_is_stereo || _coding_mode == kCodingFull) {`
`    _first_effective_channel = 0;`
`    _last_effective_channel = _num_stream_channels - 1;`
`  }`
`  else {`
`    // if the number of stream channels is 2, and the`
`    // coding mode is either "left only" or "right only"`
`    if ( _coding_mode == kCodingLeft ) {`
`      // only left`
`      _first_effective_channel = 0;`
`    }`
`    else {`
`      _first_effective_channel = 1;`
`    }`
`    _last_effective_channel = _first_effective_channel;`
`  }`
`}`

9.9.3 read_long_term_synthesis

The function *read_long_term_sysnthesis()*, shown in Table 9-41, reads the long term synthesis parameters from the bitstream. If long term synthesis is disabled, the *_lts_lag* value and the three *_lts_coefficients* are all set to zero.

Table 9-41: read_long_term_synthesis

Syntax	Reference

`void read_long_term_synthesis() {`
 	
`  const uint LTS_LAG_ALPHA = 1020;`
`  const uint LTS_LAG_MIN = 3;`
`  const uint LTS_LAG_MAX = 1022;`
`  const uint LTS_FILTER_ALPHA = 29;`
`  const uint LTS_LAG_KPARAM = 4;`
`  const uint LTS_LAG_LIMIT = 127;`
`  const float LTS_FILTER_CODEBOOK[29][3];`
`  const uint LTS_LAG_DEFAULT = 0;`
`  const float LTS_COEFFICIENTS_DEFAULT = {0.0,0.0,0.0};`
`  _lts_lag = LTS_LAG_DEFAULT;`
`  _lts_coefficients = LTS_COEFFICIENTS_DEFAULT;`
`  bool is_predictive = (_streamset->_header).is_predictive;`
`  if (lts_enabled) {`
`    bool lts_predictive = ((_previous) &&`
`                           ((_previous->_lts_lag) > LTS_LAG_DEFAULT) &&`
`                                                                (is_predictive));`
`    if(!lts_predictive) {`
	10.3.1
	
	8.2.7

Syntax	Reference
`      _lts_lag = ReadUint(LTS_LAG_ALPHA) + LTS_LAG_MIN;`	8.2.8
`      _lts_coeffients = LTS_FILTER_CODEBOOK[ReadUniform(LTS_FILTER_ALPHA)];`	8.2.11
`    }`	
`    else {`	
`      // differential lag`	
`      int prev_lts_lag = _previous->_lts_lag;`	
`      uint kparam = LTS_LAG_KPARAM;`	
`      uint alpha = LTS_LAG_ALPHA;`	
`      uint limit = LTS_LAG_LIMIT;`	
`      int diff_lts_lag = ReadGolombLimited(alpha,kparam,limit);`	8.2.13
`      diff_lts_lag = NegativeRiceMapDecode(diff_lts_lag);`	8.3.3
`      _lts_lag = Modular(prev_lts_lag + diff_lts_lag, LTS_LAG_MIN, LTS_LAG_MAX);`	8.3.2
`      // coefficients`	
`      bool changed = ReadBool();`	8.2.7
`      if (changed) {`	
`        _lts_coeffients = LTS_FILTER_CODEBOOK[ReadUniform(LTS_FILTER_ALPHA)];`	8.2.11
`      }`	
`      else {`	
`        _lts_coeffients = _previous->_lts_coeffients;`	
`      }`	
`    }`	
`  }`	
`}`	

9.9.4 read_vq_resolution_mode

The function *read_vq_resolution_mode()*, shown in Table 9-42, sets the *_is_highres* flag. If the flag is *true*, the vector quantizer used to quantize and encode the bands' spectral shapes operates in a higher resolution mode (up to 4 levels of recursive subdivision, pyramid vectors L_1 norm *maxK* = 256). If the flag is *false*, the vector quantizer operates in normal resolution mode (up to 3 levels of recursive subdivision, pyramid vectors L_1 norm *maxK* = 128).

Table 9-42: read_vq_resolution_mode

Syntax
`// read vq_resolution_mode`
`void read_vq_resolution_mode() {`
`  uint HIGHRES_MODE_BIT_THRESHOLD = 1360;`
`  if ( _total_bits < HIGHRES_MODE_BIT_THRESHOLD*_num_effective_channels ) {`
`  {`
`    is_highres = false;`
`  }`
`  else {`
`    is_highres = ReadBool();`
`  }`
`}`

9.9.5 read_mdct_mode

The function *read_mdct_mode()*, shown in Table 9-43, sets the *_is_short* flag as well as the number of MDCT blocks *_num_mdct_block* for the stream. This number is 1 when *_is_short* equals *false*, and 8 when *_is_short* equals *true.*

Table 9-43: read_mdct_mode

Syntax
`// read mdct_mode`
`uint read_mdct_mode() {`
`  uint NUM_MDCT_BLOCKS_SHORT = 8;`
`  _is_short = ReadBool();`
`  _num_mdct_block = 1;`
`  if (_is_short) {`
`    num_mdct_blocks = NUM_MDCT_BLOCKS_SHORT;`
`  }`
`}`

9.9.6 read_shf_enabled

The function *read_shf_enabled()* in Table 9-44 reads the *_shf_enabled* flag from the bit_stream.

Table 9-44: read_shf_enabled

Syntax	Reference

`// read shf enabled flag`
 	
`void read_shf_enabled() {`
`  _shf_enabled = ReadBool();`
`}`
	8.2.7

9.9.7 read_thf_enabled

The function *read_thf_enabled()*, in Table 9-45, reads the *_thf_enabled* flag from the bit_stream. This function is only called when *_is_short* equals *true*.

Table 9-45: read_thf_enabled

Syntax	Reference

`// read thf enabled flag`
 	
`void read_thf_enabled() {`
`  _thf_enabled = ReadBool();`
`}`
	8.2.7

9.9.8 read_lognorm_primary

The function *read_lognorm_primary()*, in Table 9-46, reads a first approximation of the band lognorm values from the bitstream.

Table 9-46: read_lognorm_primary

Syntax	Reference

`// read first approximation to lognorm valuesvoid read_lognorm_primary() {`
`  const uint MAX_K_BOOST = 6;`
`  uint transform_index = _is_short ? 1 : 0;`
`  uint sync_index = (_is_predictive) ? 0 : 1;`
`  // Initial (residual) values`
`  uint kboost = ReadUnary(MAX_K_BOOST);`
 	
`  for (uint band=0;band<_num_effective_bands;band++) {`
`    GolombParams gparam =`
`      LOGNORM_GOLOMB_PARAMS[fsize_index][transform_index][sync_index][band];`
`    gparam.kparam += kboost;`
`    for (uint channel=_first_effective_channel;`
`         channel<=last_effective_channel; channel++) {`
`      _lognorm_with_offset[channel][band] =`
`         PositiveRiceMapDecode(ReadGolombWithParams(gparam));`
`    }`
`  }`
`  // Reconstructed primary values (FixedPoint)`
`  LNFP f_predictor = F_PREDICTOR_TABLE[sync_index];`
`  LNFP b_predictor = B_PREDICTOR_TABLE[sync_index];`
`  for (uint band=0;band<_num_effective_bands;band++) {`
`    f_coefficient = f_predictor[band];  // prediction from previous frame`
`    b_coefficient = b_predictor[band];  // prediction from previous band`
`    LNFP pred_residual_f[_num_stream_channels];`
`    LNFP prev_band_term[_num_stream_channels];`
`    set(pred_residual_f,0);`
`    set(prev_band_term,0);`
`    for (uint channel = _first_effective_channel;`
`         channel <= _last_effective_channel; channel++) {`
`      const LNFP PREV_FRAME_FLOOR = -16.0;`
`      prev_band_term[channel] += b_coefficient * pred_residual_f[channel];`
`      pred_residual_f[channel] = _lognorm_with_offset[channel][band];`
`      LNFP prev_frame = _previous->_lognorm_with_offset[channel][band];`
	10.4.2
	
	10.4.3
	10.4.4

Syntax	Reference

`      if (prev_frame < PREV_FRAME_FLOOR) {`
`        prev_frame = PREV_FRAME_FLOOR;`
`      }`
`      _lognorm_with_offset[channel][band] = f_coefficient*prev_frame + prev_band_term +`
 	
`pred_residual_f[channel];`
`    }`
`  }`
`  transfer_lognorms();`
`}`
	9.9.15.3

9.9.9 read_num_band_blocks

The function *read_num_band_blocks()*, in Table 9-47, retrieves the number of blocks *_num_band_blocks* for each of the spectral bands as encoded in the bitstream.

Table 9-47: read_num_band_blocks

Syntax	Reference

`// read number of blocks per band`
 	
`void read_num_band_blocks () {`
`  const uint index_to_dk_pair[4] = { 2, 3, 0, 1 };`
`  const uint ALPHA_WIDE = 5;`
 	
`  int changes[NUM_ACE_BANDS]; // vector of changes to default logs of block numbers`
`  set(changes, 0); // initially all-zero`
 	
`  // read the vector of changes`
`  bool has_changes = ReadBool(); // read flag indicating whether the vector is nonzero`
	8.2.7

`  if ( has_changes ) {`
`    // vector of changes is nonzero: read and decode it`
 	
`    // read a pair (diff, k), where diff and k are in {0, 1}`
`    // diff = 1 indicates the vector of changes was encoded differentially`
`    // k indicates the encoding used for change values: 0 = unary, 1 = uniform.`
`    uint dk = ReadUnary(4); // read a unary-coded index in { 0, 1, 2, 3 }`
	8.2.10

`    dk = index_to_dk_pair[dk];  // map to a (diff, k) pair`
`    uint diff = dk >> 1; // get MS bit`
`    uint k = dk & 1;     // get LS bit`
 	
`    // The vector of changes consists of two parts: the "tail" is its longest all-zero`
`    // suffix; the "head" is the remaining prefix. Since the vector is nonzero, the`
`    // length of the head is positive. Nonzero change values belong in different`
`    // alphabets, which may differ in both size and range  depending on whether the`
`    // MDCT mode is "short" or "long", on the band  number (whether the band is`
`    // "narrow" or "wide"), and on whether differential encoding was used.`
`    // The appropriate alphabet range will be determined in each case.`
`    // read the head length (head_length-1 is uniformly coded)`
`    uint head_length = 1 + ReadUniform(_num_effective_bands);`
	8.2.11

`    // decode head symbols`
`    uint num_narrow_bands = _config -> num_narrow_bands; // number of "narrow" bands`
`    for ( uint band = 0; band < head_length; band++ ) {`
`      bool is_narrow = (band < num_narrow_bands);`
`      uint alpha = is_narrow ? (ALPHA_WIDE-1) : ALPHA_WIDE; // alphabet size`
`      uint msymb;  // mapped change value (nonnegative in [0..alpha) )`
`      if ( k == 0 ) {`
`        // decode a unary-coded symbol`
`        msymb = ReadUnary(alpha);`
	8.2.10

`      }`
`      else {`
`        // decode a uniform symbol`
`        msymb = ReadUniform(alpha);`
	8.2.11

`      }`
`      // map to a signed change value`
`      changes[band] = map_tf_change(msymb, ALPHA_WIDE, is_narrow, diff);`
	9.9.10.1

`    }  // loop on band`
`    if ( diff ) {`
`      // undo differential encoding`
`      undo_differential(changes, ALPHA_WIDE);`
	9.9.10.2

`    }`
`  }  // if ( has_changes )`
`  // apply changes and convert to number of blocks`	

Syntax	Reference

uint base_log = _is_short ? 3 : 0;
for (uint band=0; band < _num_effective_bands; band++) {
uint the_log = base_log + changes[band];
_num_band_blocks[band] == (1 << the_log);
}
}	

9.9.10 Functions Supporting read_num_band_blocks

9.9.10.1 map_tf_change

Auxiliary function *map_tf_change()* maps a decoded unsigned symbol to a signed change value.

Table 9-48: map_tf_change

Syntax	Reference

// map_tf_change
 	
int map_tf_change(uint msymb, uint alpha_base, bool is_narrow, uint diff) {
const int map_short_narrow[] = { 0, -1, -2, -3 };
const int map_short_wide[] = { 0, 1, -1, -2, -3 };
 	
uint alpha = is_narrow ? (alpha_base-1) : alpha_base; // alphabet size
int symb;
if (diff == 0) {
// non-differential
if (_is_short) {
if (is_narrow) {
symb = map_short_narrow[msymb];
}
else {
symb = map_short_wide[msymb];
}
}
else {
symb = (int)msymb;
}
}
else {
// differential: further processing may be needed after undoing differentiation
symb = PositiveRiceMap(msymb);
}
return symb;
}
	8.3.4

9.9.10.2 undo_differential

The function *undo_differential()* reverses differentiation in a differentially encoded vector of integers.

Table 9-49: undo_differential

Syntax	Reference

// undo differentiation
 	
undo_differential(int &vec[], uint alpha_base)
{
uint num_narrow_bands = _config -> num_narrow_bands;
enforce_range(vec[0], alpha_base, num_narrow_bands > 0);
for (uint band=1; band < _num_effective_bands; band++) {
vec[band] = vec[band] + vec[band-1];
enforce_range(vec[band], alpha_base, band < num_narrow_bands);
}
}
	9.9.10.3
	
	9.9.10.3

9.9.10.3 enforce_range

Table 9-50: enforce_range

Syntax	Reference

`// enforce range`
 	
`void enforce_range(int &val, uint alpha_base, bool is_narrow)`
`{`
`  const int LONGBLOCK_MIN = 0;`
`  const int SHORTBLOCK_MIN = -3;`
`  uint alpha = is_narrow ? (alpha_base-1) : alpha_base;`
`  int minsymb = _is_short ? SHORTBLOCK_MIN : LONGBLOCK_MIN;`
`  // bring to range [minsymb, minsymb+alpha) modulo alpha`
`  val = Modular(val, minsymb, minsymb + alpha - 1);`
`}`
	8.3.2

9.9.11 read_conditioning_level

The function *read_reconditioning_level()* retrieves the reconditioning level from the bitstream.

Table 9-51: read_reconditioning_level

Syntax	Reference

`// read reconditioning level`
 	
`read_reconditioning_level() {`
`  reconditioning_level = ReadUint(2);`
`}`
	8.2.8

9.9.12 read_allocation

The function *read_allocation()* assigns to each band a bit budget, determining the quality of reconstruction for each band. The parameter *_num_coded_bands* indicates the region of bands that have spectral components encoded in the bitstream. Bands that have no encoded spectral components are reconstructed based on lognorm values only. Several models are allowed to compute band allocations. A first step in *read_allocation()* is to read the allocation model. Some allocation models need additional parameters that are subsequently read from the bitstream.

Table 9-52: read_allocation

Syntax	Reference

`// Retrieve band allocation data from the bitstream`
 	
`void read_allocation() {`
`  // initialize adjustments`
`  set(_alloc_delta,0);`
`  set(_alloc,0);`
`  // read number of coded bands`
`  read_num_coded_bands();`
`  // number of bits left for stream`
`  uint remaining_bits = BitsLeft();`
`  // bail if there is nothing there`
`  if (_num_coded_bands == 0) {`
`    if (remaining_bits >= _num_effective_bands * _min_guaranteed_alloc) {`
`      for (int i=0;i<_num_effective_bands;i++) {`
`        _alloc[band] = _min_guaranteed_alloc;`
`      }`
`    }`
`    return;`
`  }`
`  // retrieve number of ms_mono bands`
`  read_num_ms_mono_bands();`
`  // retrieve stereo flags`
`  read_stereo_flags();`
`  read_alloc_model();`
`  // Read allocation model data`
`  bool choose_basic = false;`
`  switch(_alloc_model) {`
	9.9.13.1
	8.2.4
	
	9.9.13.7
	9.9.13.8
	9.9.13.3

Syntax	Reference
`  case kUnsignedDeltaFromBasic: {`	
`    read_alloc_model_parameter();`	9.9.13.4
`    read_alloc_model_delta();`	9.9.13.5
`    break;`	
`  }`	
`  case kUnsignedModifiedDeltaFromPrimaryLSQ:`	
`  case kUnsignedDeltaFromPrimaryLSQ: {`	
`    read_alloc_model_parameter();`	9.9.13.4
`    read_alloc_model_delta();`	9.9.13.5
`    break;`	
`  }`	
`  case kSignedDeltaFromPrimaryLSQ: {`	
`    read_alloc_model_parameter();`	9.9.13.4
`    break;`	
`  }`	
`  case: kSignedDeltaFromPreviousFrame: {`	
`    // this mode does not read a model parameter from the bitstream;`	
`    // the parameter is set to a default value`	
`    _alloc_model_parameter = ALLOC_MODEL_PARAMETER_DEFAULT;`	
`    break;`	
`  }`	
`  case kSignedDiffFromFoundation: {`	
`    choose_basic = ReadBool();`	8.2.7
`    if (choose_basic) {`	
`      read_alloc_model_parameter();`	9.9.13.4
`    }`	
`    break;`	
`  }`	
`  default: {`	
`    abort("invalid mode");`	
`  }`	
`}`	
`uint adjustments[NUM_ACE_BAND]; // auxiliary array for intermediate adjustments`	
`// Derive bit allocations depending on the mode`	
`switch (_alloc_model) {`	
`  // Allocation bits-per-band derived from`	
`  // unsigned adjustments to a basic curve.`	
`  case: kUnsignedDeltaFromBasic: {`	
`    for (uint band = 0; band < num_effective_bands; band++) {`	
`      adjustments[band] = alloc_delta[band];`	
`    }`	
`    do_allocation_model(remaining_bits, adjustments, _alloc);`	9.9.13.9
`    break;`	
`  }`	
`  // Allocation bits-per-band derived from lognorms`	
`  case: kUnsignedModifiedDeltaFromPrimaryLSQ: {`	
`    compute_adjustments_from_lognorms (adjustments);`	9.9.13.11
`    for (uint band = 0; band < _num_effective_bands; band++) {`	
`      if (adjustments[band] > num_bits_step(band)) {`	9.9.13.6
`        adjustments[band] -= num_bits_step(band);`	9.9.13.6
`      }`	
`      adjustments[band] += _alloc_delta[band];`	
`    }`	
`    do_allocation_model(remaining_bits, adjustments, _alloc);`	9.9.13.9
`    break;`	
`  }`	
`  // Allocation bits-per-band derived from unsigned adjustments`	
`  // to a curve computed from primary quantized lognorm values`	
`  // (standard interpretation of adjustments).`	
`  case: kUnsignedDeltaFromPrimaryLognorm: {`	
`    compute_adjustments_from_lognorm(adjustments);`	9.9.13.11
`    for (uint band = 0; band < _num_effective_bands; band++) {`	
`      adjustments[band] += alloc_delta[band];`	
`      if (adjustment[band] < 0) {`	
`        adjustment[band] = 0;`	
`      }`	
`    }`	
`    // populate allocation vector`	
`    do_allocation_model(num_bits_available, adjustments, _alloc);`	9.9.13.9
`    break;`	
`  }`	
`  // Allocation bits-per-band derived from signed adjustments`	
`  // to a curve computed from primary quantized lognorm values`	
`  // (standard interpretation of adjustments).`	
`  case: kSignedDeltaFromPrimaryLognorm: {`	
`    uint alloc_ref[NUM_ACE_BANDS]; // will store the reference allocations`	

Syntax	Reference
compute_adjustments_from_lognorms(adjustments);	9.9.13.11
// estimate the code length of the allocation description	
uint est_cost = estimate_cost_from_adjustments(_num_coded_bands, adjustments);	9.9.13.12
// construct reference allocation	
do_allocation_model(num_bits_available-est_cost, adjustments, alloc_ref);	9.9.13.9
read_alloc_diff_steps(_num_coded_bands, alloc_ref, _min_guaranteed_alloc, _num_coded_bands, _alloc);	9.9.13.13
break;	
}	
// Derive allocation bits-per-band	
// with respect to previous frame	
case: kSignedDeltaFromPreviousFrame: {	
// read the allocation vector using the previous frame allocation	
// as reference	
uint alloc_ref[NUM_ACE_BANDS];	
uint n_bands_ref, ref_guaranteed_alloc;	
if (!_is_predictive) {	
set(alloc_ref[], 0);	
n_bands_ref = 0;	
ref_guaranteed_alloc = 0;	
}	
else {	
alloc_ref = _previous->_alloc;	
n_bands_ref = _previous->_num_coded_bands;	
ref_guaranteed_alloc = _previous->_min_guaranteed_alloc;	
}	
read_alloc_diff_steps(n_bands_ref, alloc_ref, ref_guaranteed_alloc, _num_coded_bands, _alloc);	9.9.13.13
break;	
}	
// Differentially code allocation bits-per-band with respect	
// to a Foundation Curve	
case: kSignedDiffFromFoundation: {	
const uint cost_estimate[2] = { 60, 72 };	
int zeros[NUM_ACE_BANDS];	
int foundation[NUM_ACE_BANDS];	
// get a rough estimate of the cost of the encoding	
uint est_cost = cost_estimate[_num_effective_channels - 1];	
// pass zero adjustments to get basic foundation	
set(zeros,0);	
if (choose_basic) {	
do_allocation_model(num_bits_available - est_cost, zeros, foundation);	9.9.13.9
}	
// populate allocation vector	
compute_alloc_from_foundation(choose_basic, foundation);	9.9.13.15
break;	
}	
default: { abort("invalid code")); }	
}	
// Compute first lognorm refinement	
compute_first_lognorm_refinement_alloc();	9.9.13.16
// compute total allocation	
uint total_allocs = 0;	
for (uint band = 0; band < _num_effective_bands; band++) {	
total_allocs += _alloc[bnd];	
total_allocs += _num_effective_channels * _first_refinement_alloc[bnd];	
}	
// record unallocated bits	
_unallocated_bits = BitsTotal() - total_allocs;	8.2.5
}	

9.9.13 Functions use by read_allocation

9.9.13.1 read_num_coded_bands

The function *read_num_coded_bands()* reads the number of coded bands. If no bands are coded, the function *read_allocation* returns immediately.

Table 9-53: read_num_coded_bands

Syntax	Reference

`// read the number of coded bands (>= 0, <= _num_effective_bands)`
`void read_num_coded_bands() {`
 	
`  const uint CODED_BANDS_GOLOMB_KPARAM = 0;`
`  const uint CODED_BANDS_GOLOMBS_LIMIT = 7;`
 	
`  GolombParams gparams;`
`  gparams.alpha = _num_effective_bands + 1;`
`  gparams.kparam = CODED_BANDS_GOLOMB_KPARAM;`
`  gparams.limit = CODED_BANDS_GOLOMBS_LIMIT;`
 	
`  int prev_noncoded = _num_effective_bands - _previous->_num_coded_bands;`
`  int num_noncoded_bands = read_num_bands(gparams, prev_noncoded);`
`  _num_coded_bands = _num_effective_bands - num_noncoded_bands;`
`}`
	9.9.13.2

9.9.13.2 read_num_bands

The function *read_num_bands()* reads the cardinality of a sequence of bands.

Table 9-54: read_num_bands

Syntax	Reference

`// read a cardinality of sequence of bands`
 	
`uint read_num_bands(GolombParams gparams, uint prev_num_bands) {`
 	
`  if (!_is_predictive) {`
`    num_bands = ReadUniform(gparams.alpha);`
`  }`
`  else {`
`    int mapped_residual = (int) ReadGolombWithParams(gparams);`
`    int residual = NegativeMapRice(mapped_residual);`
`    residual += prev_num bands;`
`    num_bands = (uint)Modular(residual, 0, gparams.alpha-1);`
`  }`
 	
`  return num_bands;`
`}`
	8.2.11
	
	8.2.14
	8.3.3
	
	8.3.2

9.9.13.3 read_allocation_model

The function *read_allocation_model()* reads the allocation model from the bitstream.

Table 9-55: read_alloc_mode

Syntax	Reference

`// Read the allocation model from the bitstream`
 	
`void read_alloc_mode() {`
 	
`  const uint BIT_ALLOCATION_SIGNALLING_MODE_ALPHA = 6;`
 	
`  _alloc_model = ReadUniform(BIT_ALLOCATION_SIGNALLING_MODE_ALPHA);`
`}`
	8.2.11

9.9.13.4 read_alloc_model_parameter

The function *read_alloc_model_parameter()* reads the allocation model parameter from the bitstream.

Table 9-56: read_alloc_model_parameter

Syntax	Reference
void read_alloc_model_parameter() {	
const uint ALLOC_MODEL_PARAM_ALPHA = 11;	
if (!_is_predictive) {	
_alloc_model_parameter = ReadUnary(ALLOC_MODEL_PARAM_ALPHA);	8.2.10
}	
else {	
uint prev_alloc_model_parameter = _previous->_alloc_model_parameter;	
uint diff_alloc_model_parameter = ReadUniform(ALLOC_MODEL_PARAM_ALPHA);	8.2.11
_alloc_model_parameter = DecodeCenterMap(ALLOC_MODEL_PARAM_ALPHA, prev_alloc_model_parameter, diff_alloc_model_parameter);	8.3.5
}	
}	

9.9.13.5 read_alloc_model_delta

The function *read_alloc_model_delta()* reads the adjustment to a reference bit allocation. The reference allocation is computed from previous and/or current stream data. The encoding of adjustments depends on the allocation model.

Table 9-57: read_alloc_model_delta

Syntax	Reference
// read adjustments to reference model (model dependent)	
void read_alloc_model_delta() {	
uint head_length = 0;	
int steps = 0;	
set(_alloc_delta[], 0); // clear the vector of deltas	
switch (_alloc_model) {	
case kUnsignedDeltaFromBasic:	
case kUnsignedModifiedDeltaFromPrimaryLSQ:	
// read unsigned values	
head_length = ReadUniform(_num_effective_bands+1);	8.2.11
break;	
case kUnsignedDeltaFromPrimaryLSQ:	
// read unsigned values (zero preferred)	
bool all_zero = ReadBool();	8.2.7
if (!all_zero) {	
head_length = ReadUniform(_num_effective_bands) + 1;	8.2.11
}	
else {	
head_length = 0;	
}	
break;	
default:	
abort("invalid mode");	
}	
for (uint band=0;band<head_length;band++) {	
steps = ReadUnary(num_steps_band(band)+1);	8.2.11
if ((steps > 0) && (_alloc_model == kSignedDeltaFromPrimaryLSQ)) {	
bool sign = ReadBool();	8.2.7
if (sign) {	
steps = -steps;	
}	
}	
_alloc_delta[band] = steps * num_bits_step(band);	9.9.13.6
}	
}	

9.9.13.6 num_bits_step

The function *num_bits_step()* computes the number of bits per quantization step as a function of the band index.

Table 9-58: num_bits_step

Syntax

uint num_bits_step(uint band) {
 	
  uint num_bits = 0;
 	
  const uint ADJUSTMENT_STEP_SMALL_VALUE = 6;
  const uint ADJUSTMENT_STEP_LARGE_THRESH = 48;
  const uint ADJUSTMENT_STEP_LARGE_SCALE_FACTOR = 8;
 	
  uint num_bins = _num_effective_channels * _config->bandsize[band];
 	
  // piecewise linear
  uint num_bits = 0;
 	
  // small num_bins
  if (num_bins < ADJUSTMENT_STEP_SMALL_VALUE) {
  num_bits = num_bins;
  // medium num_bins
  } else if (num_bins < ADJUSTMENT_STEP_LARGE_THRESH) {
  num_bits = ADJUSTMENT_STEP_SMALL_VALUE;
  // large num_bins
  } else {
  num_bits = num_bins / ADJUSTMENT_STEP_LARGE_SCALE_FACTOR;
  }
 	
  return num_bits;
}

9.9.13.7 read_num_ms_mono_bands

The function *read_num_ms_mono_bands()* reads the number of midside-mono bands. A band is called midside-mono when only the mid or side band is encoded, and the other band is set to zero. The highest bands in a stereo stream could be encoded as midside-mono.

Table 9-59: read_num_ms_mono_bands

Syntax	Reference

// retrieve the number of ms_mono bands.
 	
void read_num_ms_mono_bands() {
 	
  const uint MS_BANDS_GOLOMB_KPARAM = 0;
  const uint MS_BANDS_GOLOMBS_LIMIT = 7;
 	
  GolombParams gparams;
  gparams.alpha = _num_effective_bands;
  gparams.kparam = MS_BANDS_BANDS_GOLOMB_KPARAM;
  gparams.limit = MS_BANDS_GOLOMBS_LIMIT;
  int prev_ms_mono_bands = _previous->_num_ms_mono_bands;
 	
  _num_ms_mono_bands = read_num_bands(gparams, prev_msmono_bands);
 	
}
	9.9.13.2

9.9.13.8 read_stereo_flags

The function *read_stereo_flags()* reads stereo information from the bitstream, specifying the stereo coding mode for each of the coded bands in the stream.

Table 9-60: read_stereo_flags

Syntax	Reference

// read stereo flags from the bitstream
 	
void read_stereo_flags() {
 	
  const uint LOW_REGION_THRESHOLD = 4; // Number of bands in first region
  const uint MID_REGION_THRESHOLD = 12; // Number of bands in first and	

Syntax	Reference

// second region.
uint LR_STEREO_NUM_BANDS_PER_FREQ_REGION[3] = {4,8,_num_effective_bands-12};
 	
// return if not stereo
if (!_is_stereo) {
  return;
}
 	
uint num_non_ms_mono_bands = _num_effective_bands - _num_ms_mono_bands;
 	
// set the high bands to midside mono
for (uint band = num_non_ms_mono_bands;band<_num_effective_bands;band++) {
  _stereo_flag[band] = kStereoMidSideMono;
}
 	
// Read additional flags if both channels are coded
if (_coding_mode == kCodingFull) {
 	
  bool has_lr_stereo = false;
 	
  // full frame stereo flag
  _full_frame_stereo_coding_flag = ReadBool();
	8.2.7

  // decision for full frame
  bool frame_ms_flag;
 	
  if (_full_frame_stereo_coding_flag) {
  frame_ms_flag = ReadBool();
	8.2.7

  for (int band=0;band<num_ms_stereo_bands;band++) {
  _stereo_flag[band] = frame_ms_flag ? kStereoMidSide : kStereoLeftRight;
  }
  if (!frame_ms_flag) {
  has_lr_stereo = true;
  }
  }
  else {
  for (int band=0;band<num_ms_stereo_bands;band++) {
  bool band_ms_flag = ReadBool();
  _stereo_flag[band] = band_ms_flag ? kStereoMidSide : kStereoLeftRight;
  if (!band_ms_flag) {
  has_lr_stereo = true;
  }
  }
  }
 	
  // LR allocation regions
  uint num_lr_alloc_regions;
  if (num_non_ms_mono_bands == 0) {
  num_lr_alloc_regions =0;
  }
  else {
  num_lr_alloc_regions = compute_region(num_non_ms_mono_bands - 1) + 1;
	9.10.3.5.2

  }
 	
  // If LR may appear, read stereo allocation per region
  if (has_lr_stereo) {
 	
  bool all_5050 = ReadBool();
  if (all_5050) {
  for (uint region = 0; region < num_lr_alloc_regions;region++) {
  _lr_alloc_region_mode[region] = kLrStereoAllocation5050;
  }
  }
  else {
  uint band = 0;
  for (uint region = 0; region < _num_lr_alloc_regions;region++) {
  bool region_includes_lr_stereo_band = false;
  for (uint j = 0; j < LR_STEREO_NUM_BANDS_PER_FREQ_REGION[i]; j++) {
  if (_stereo_flag(band) == kStereoLeftRight)
  region_includes_lr_stereo_band = true;
  band++;
  }
  if (region_includes_lr_stereo_band) {
  uint mode = ReadUint(1);
  if (mode) {
  mode = ReadUint(2) + 1;	

Syntax	Reference

}
_lr_alloc_region_mode[region] = mode;
}
}
}
}
}
}	

9.9.13.9 do_allocation_model

The function *do_allocation_model()* iteratively searches over the allocation model space until the target allocation is met. The actual model calculations are done by the sub-function *do_allocation_model_step*. The function returns the number of total number of allocated bits.

Table 9-61: do_allocation_model

Syntax	Reference

// iteratively search the allocation model space until
// a target bit budget is met.
// parameter[in] num_available_bits: target bit budget
// parameter[in] adjustments: per-band adjustments to apply to basic curve
// parameter[out] alloc_out: computed allocation
// return: total number of bits allocated.
//
// The search is over a space of basic allocation curves that, when added to the
// (curve of) adjustments results in an output allocation curve.
 	
uint do_allocation_model(uint num_available_bits, int adjustments[], uint &alloc_out[])
{
 	
const BAFP ALLOCATION_CURVE_SEARCH_RANGE = 8.0
const uint ALLOCATION_CURVE_SEARCH_RANGE_LOG = 13;
 	
BAFP low = 0;
BAFP high = ALLOCATION_CURVE_SEARCH_RANGE;
 	
for (uint i = 0; i < ALLOCATION_CURVE_SEARCH_RANGE_LOG; i++){
BAFP mid = (low + high)/2;
uint num_allocated_bits = do_allocation_model_step(mid, adjustments, alloc_out);
if (num_allocated_bits > num_available_bits) {
high = mid;
}
else {
low = mid;
}
}
 	
return do_allocation_model_step(num_available_bits, low, adjustments, alloc_out);
}
	9.9.13.10

9.9.13.10 do_allocation_model_step

The function *do_allocation_model_step()* computes a per-band bit allocation given a bit allocation for the first bin, a set of adjustment values (read or computed earlier) and an allocation model parameter (read earlier from the bitstream). The function returns the total number of bits in the allocation.

Table 9-62: do_allocation_model_step

Syntax

// compute a bit allocation per band for all bands, given a
// bit allocation for the first bin, and a vector of adjustments.
//
// parameter[in] first_bin_bits: number of bits allocated to first bin
// parameter[in] adjustments: adjustments to model.
// parameter[out] alloc_out: the computed allocation
// return: total number of bits allocated.
 	
uint do_allocation_model_step(BAFP first_bin_bits, int adjustments[], uint &alloc_out[])

Syntax

```
{
  const BAFP BAFACTOR[21] = {
    0.080078125, 0.087890625, 0.095703125, 0.103515625,
    0.111328125, 0.119140625, 0.126953125, 0.134765625,
    0.142578125, 0.150390625, 0.158203125, 0.166015625,
    0.173828125, 0.181640625, 0.189453125, 0.197265625,
    0.205078125, 0.212890625, 0.220703125, 0.228515625,
    0.236328125
  };

  const uint HIGH_BANDS_THRESHOLD = 19;
  const uint HIGH_BANDS_CORRECTION = 1;

  // Initialize allocation
  set(alloc_out[], 0); // start with a clean allocation vector

  // Compute base allocations
  for (uint band = 0; band < _num_coded_bands; band++) {

    // band size
    uint num_band_bins = _config->bandsize[band];

    // Per bin allocations
    BAFP bits_per_bin = first_bin_bits;
    bits_per_bin -= band * BAFACTOR[_alloc_model_parameter];
    bits_per_bin -= (band > HIGH_BANDS_THRESHOLD) ? HIGH_BANDS_CORRECTION : 0;
    bits_per_bin = max(bits_per_bin,0);

    // Per band allocations
    BAFP bits_per_band = num_band_bins * _num_stream_channels * bits_per_bin;
    bits_per_band += (first_bin_bits > 0) ? adjustments[band] : 0;
    bits_per_band = max(_min_guaranteed_alloc, bits_per_band);
    bits_per_band = min(_alloc_max[band], bits_per_band);

    // Assigning to allocation array
    alloc_out[band] = bits_per_band;
  }

  // Find last band with allocation leq than minimum
  int last_coded_band = -1;
  for (int band = _num_coded_bands - 1; band >= 0; band--) {
    if (alloc_out[band] >= _alloc_min[band]) {
      last_coded_band = band;
      break;
    }
  }

  // All bands up to the last_coded_band get at least the minimum usable for VQ
  for (int band = last_coded_band; band >= 0; band--) {
    alloc_out[band] = max(alloc_out[band], _alloc_min[band]);
  }

  // All bands after last coded band just get the guaranteed minimum
  for (int band = last_coded_band + 1; band < _num_coded_bands; band++) {
    alloc_out[band] = _min_guaranteed_alloc;
  }

  // Compute total bit allocation
  uint allocated_bits = 0;
  for (band = 0; band < _num_effective_bands; band++) {
    allocated_bits += alloc_out[band];
  }

  // Return total bit consumption
  return allocated_bits;
}
```

9.9.13.11 compute_adjustments_from_lognorms

The function *compute_adjustments_from_lognorms()* computes adjustments from primary lognorm values. This function depends on the unmasked value levels computed by the function *compute_unmasked_levels*.

Table 9-63: compute_adjustments_from_lognorms

Syntax	Reference

`// compute adjustments from quantized primary lognorm values`
`//`
`// parameter[out] adjustments: adjustment values`
 	
`void compute_adjustments_from_lognorms (uint &adjustments[]) {`
 	
`  LNFP unmasked_levels[NUM_ACE_BANDS];`
 	
`  // compute unmasked levels`
`  compute_unmasked_levels(unmasked_levels);`
 	
`  // total adjustment expenditure`
`  int adjust_total = 0;`
 	
`  for (uint band = 0; band < _num_effective_bands; band++) {`
 	
`    int num_band_bins = num_stream_channels * _config->bandsize[band];`
`    int num_bits_adjust = unmasked_level[band] * num_band_bins;`
`    num_bits_adjust = min(num_bits_adjust, _alloc_max[band]);`
 	
`    adjustments[band] = num_bits_adjust;`
`    adjust_total += num_bits_adjust;`
`  }`
 	
`  // Total for stream`
`  uint total_bits = BitsTotal();`
 	
`  // cap adjustments to <= half total available bits`
`  const uint ACE_NUM_ESTIMATE_NONVQ_BITS = 60;`
`  int cap = max((total_bits - ACE_NUM_ESTIMATE_NONVQ_BITS)/2, 0);`
`  if ( adjust_total > cap ) {`
`    // pro-rate each adjustment to guarantee the total is below the cap`
`    for ( uint band=0; band < _num_effective_bands; band++ ) {`
`      if (adjustments[band] > 0 ) {`
`        adjustments[band] = (adjustments[band] * cap)/adjust_total;`
`      }`
`    }`
`  }`
`}`
	8.2.5

9.9.13.12 estimate_cost_from_adjustments

The function *estimate_cost_from_adjustments()* estimates length of encoding from a vector of unsigned adjustments.

Table 9-64: estimate_cost_from_adjustments

Syntax	Reference

`// estimate cost from a vector of adjustments`
`uint estimate_cost_from_adjustments(uint n_bands, uint adjustments[])`
`{`
`  uint cost = 0;`
`  for ( uint band=0; band < n_bands; band++ ) {`
`    uint stepsize = num_bits_step(band);`
`    cost += 1;`
`    if ( adjustments[band] >  0 ) {`
`      cost += (adjustments[band] + stepsize/2)/stepsize;`
`    }`
`  }`
`  return cost;`
`}`
	9.9.13.6

9.9.13.13 read_alloc_diff_steps

The function *read_alloc_diff_steps()* computes the band allocation for the current frame as a differential with respect to a reference allocation. Quantized residual values are read from the bitstream and added to the values of the reference allocation.

Table 9-65: read_alloc_diff_steps

Syntax	Reference
`// Compute current frame bit allocation by applying adjustments to a reference`	
`// bit allocation. Adjustments are decoded as quantized deltas;`	
`// they are unquantized and added to the reference allocation.`	
`void read_alloc_diff_steps (uint n_bands_ref, uint alloc_ref[],`	
`  uint ref_guaranteed_alloc,`	
`  uint n_bands, uint &alloc[]) {`	
`  // arrays to hold the possible range of delta for each band`	
`  int delta_min[NUM_ACE_BANDS], delta_max[NUM_ACE_BANDS];`	
`  // arrays to hold allocation references and maxima with guaranteed bits removed`	
`  uint alloc_ref1[NUM_ACE_BANDS], allocs_max[NUM_ACE_BANDS];`	
`  // copy the reference allocation and allocation maxima after removing`	
`  // the bits guaranteed to each bands`	
`  remove_guaranteed_bits(alloc_ref, alloc_ref1, alloc_max, ref_guaranteed_alloc);`	9.9.13.14.1
`  // compute the range of delta for each band, and the range of`	
`  // the Golomb parameter kparam`	
`  uint k_alpha, kparam;`	
`  compute_delta_alphabets(alloc_ref1, alloc_max, delta_min, delta_max, k_alpha);`	9.9.13.14.2
`  // read the Golomb parameter kparam`	
`  kparam = ReadUnary(k_alpha);`	8.2.10
`  // read a value of delta for each band`	
`  for ( uint band=0; band < _num_coded_bands; band++ ) {`	
`    int alpha = delta_max[band] - delta_min[band] + 1;`	
`    int d_min = delta_min[band];`	
`    // read an unsigned, Golomb-encoded, mapped delta value`	
`    uint udelta = ReadGolomb(kparam, alpha);`	8.2.12
`    // unmap it`	
`    udelta = DecodeCenterMap(alpha, (uint)(-d_min), udelta);`	8.3.5
`    delta = (int)udelta + d_min; // undo a shift by d_min done at the encoder`	
`    // get the reference allocation`	
`    int ref = (band < n_bands_ref) ? alloc_ref1[band] : 0;`	
`    // get the quantization step size`	
`    int stepsize = num_bits_step(band);`	9.9.13.6
`    // get the upper bound on the allocation`	
`    int amax = alloc_max[band];`	
`    // tentative value; may be out of bounds`	
`    int allocation = ref + stepsize * delta;`	
`    // clamp to [0, amax]`	
`    if ( alloc < 0 )`	
`      allocation = 0;`	
`    else if ( alloc > amax )`	
`      allocation = amax;`	
`    // store the allocation`	
`    alloc[band] = allocation;`	
`  }`	
`  // the decoding reconstructed the allocations with the guaranteed bits`	
`  // removed; restore them`	
`  restore_guaranteed_bits(n_bands, alloc);`	9.9.13.14.3
`}`	

9.9.13.14 Functions Supporting read_alloc_diff_steps

9.9.13.14.1 remove_guaranteed_bits

This function removes the bits guaranteed to each band from a reference allocation and the vector of allocation maxima.

Table 9-66: remove_guaranteed_bits

Syntax
`// copy reference allocation and max allocations to new vectors after removing`
`// the bits guaranteed to each band`
`void remove_guaranteed_bits(uint alloc_ref[], uint &alloc_ref1[], uint &alloc_max[],`
`  uint ref_guaranteed_alloc)`
`{`
`  for ( uint band=0; band < _num_effective_bands; band++ ) {`
`    if ( alloc_ref[band] >= ref_guaranteed_alloc ) {`
`      alloc_ref1[band] = alloc_ref[bnd] - ref_guaranteed_alloc;`
`    }`

Syntax

`    else {`
`      alloc_ref1[band] = 0;`
`    }`
`    alloc_max = _alloc_max[band] - _min_guaranteed_alloc;`
`}`

9.9.13.14.2 compute_delta_alphabets

compute_delta_alphabets() will compute the ranges of possible values of *delta* for each band, and the range of the Golomb parameter *k* that will be used to decode the deltas.

A side effect of this function is that it may modify the reference vector *alloc_ref*.

Table 9-67: compute_delta_alphabets

Syntax	Reference

`void compute_delta_alphabets(uint &alloc_ref[], uint alloc_max[],`
`                              int &delta_min[], int &delta_max[], uint &k_alpha)`
`{`
`  set(delta_min[], 0);`
`  set(delta_max[], 0);`
`  uint nbands_ref = _is_predictive ? _previous->_num_coded_bands : 0;`
`  int max_alpha = 0;`
`  for ( uint band = 0; band < _num_coded_bands; band++ ) {`
`    int ref = (band < nbands_ref) ? alloc_ref[band] : 0;`
`    int amax = alloc_max[band];`
`    if ( ref > amax ) {`
`      ref = amax;`
`      alloc_ref[band] = ref;`
`    }`
 	
`    // get allocation quantization step for this band`
`    int stepsize = num_bits_step(band);`
	9.9.13.6

`    _delta_max[band] = (amax - ref + stepsize - 1)/stepsize;`
`    _delta_min[band] = (ref + stepsize - 1)/stepsize;`
`    alpha = _delta_max[band] - _delta_min[band] + 1;`
`    if ( alpha > max_alpha ) {`
`      max_alpha = alpha;`
`    }`
`  }`
`  k_alpha = NumBits(max_alpha-1);`
`}`
	8.3.1

9.9.13.14.3 restore_guaranteed_bits

This function restores guaranteed bits to the bit allocation vector.

Table 9-68: restore_guaranteed_bits

Syntax

`// retore guaranteed bits to the bit allocation vector`
`void restore_guaranteed_bits(uint n_bands, uint &alloc[])`
`{`
`  for ( uint band=0; band < n_bands; band++ ) {`
`    alloc [band] += _min_guaranteed_alloc;`
`  }`
`}`

9.9.13.15 compute_alloc_from_foundation

The function *compute_alloc_from_foundation()* computes the band allocation for the current frame as a differential with respect to a foundation curve. Residual values are read from the bitstream and added to the values of the foundation curve (with modular correction).

Table 9-69: compute_alloc_from_foundation

Syntax	Reference

`// compute current allocations as a differential with respect to`
`// a foundation curve.`
 	
`void compute_alloc_from_foundation (bool choose_basic, uint foundation[]) {`
 	
`  const uint MAX_BA_QUANT_LEVEL = 6;`
`  uint alloc_quant[];`
 	
`  // read quantization level`
`  uint quant_level = ReadUnary(MAX_BA_QUANT_LEVEL+1);`
`  // set _alloc to zero`
`  set(_alloc,0);`
 	
`  // determine Golomb k-parameter range`
`  uint alpha = 0;`
`  for (uint band=0;band<_num_coded_bands;band++) {`
`    alpha_quant[band] = (_alloc_max[band}>>quant_level) + 1;`
`    if (alpha_quant[band] > alpha) {`
`      alpha = alpha_quant[band];`
`    }`
`  }`
 	
`  // read Golomb k-parameter`
`  uint kparam = ReadUniform(alpha);`
 	
`  // read deltas and populate alloc`
`  if (choose_basic) {`
`    // predict from foundation`
`    for (uint band = 0;band < _num_coded_band; band++) {`
`      int delta = ReadGolomb(alpha_quant[band],kparam);`
`      delta = NegativeRiceMapDecode(delta);`
`      prev_alloc = foundation[band] / (1u << quant_level);`
`      _alloc[band] = Modular(prev_alloc + delta, 0, alpha_qant[band]-1) << quant_level;`
`    }`
`  }`
`  else {`
`    // predict from zero`
`    for (uint band = 0;band < _num_coded_band; band++) {`
`      _alloc[band] = ReadGolomb(alpha_quant[band],kparam);`
`      _alloc[band] *= (1u << quant_level);`
`    }`
`  }`
`}`
	8.2.10
	
	8.2.11
	
	8.2.12
	8.3.3
	8.3.2
	
	8.2.12

9.9.13.16 compute_first_lognorm_refinement_alloc

The function *compute_first_lognorm_refinement_alloc()*, shown in Table 9-70, computes the per-band budget for lognorm refinement.

Table 9-70: compute_first_lognorm_refinement_alloc

Syntax	Reference

`// The function compute_first_lognorm_refinement_alloc computes`
`// the budget for lognorm refinement, one value per band.`
 	
`uint compute_first_lognorm_refinement_alloc() {`
 	
`  const uint MAX_REFINEMENT_ALLOC = 8;`
`  uint carryover = 0;`
 	
`  uint bandsize[] = _config->bandsize;`
`  uint refinement_delta[] = _config->_refinement_delta;`
 	
`  for (uint band = 0; band < _num_coded_bands;band++) {`
 	
`    int L = _config->bandsize[band];`
`    int band_allocation = _alloc[band];`
`    int band_allocation_max = _alloc_max[band];`
 	
`    int total_bits = band_allocation + carryover;`
`    band_allocation = min(total_bits,band_allocation_max);`	

Syntax	Reference

uint remainder = total_bits - band_allocation;
 	
BAFP num_bins = L*_num_stream_channels;
BAFP tentative_refinement = band_allocation + num_bins*_refinement_delta[band];
 	
// Boost small values
tentative_refinement = adjust_refinement(tentative_refinement,num_bins,band);
 	
// semi-final refinement
int refinement = tentative_refinement + num_bins/2) / num_bins;
 	
// Clamp
if (refinement < 0) {
refinement = 0;
}
if (refinement * _num_stream_channels > band_allocation) {
refinement = band_allocation / _num_stream_channels;
}
if (refinement > MAX_REFINEMENT_ALLOC) {
refinement = MAX_REFINEMENT_ALLOC;
}
 	
// Refinement allocation
_first_refinement_alloc[band] = refinement;
_alloc[band] -= refinement;
 	
// Remaining bits
if (remainder > 0) {
uint wish = MAX_REFINEMENT_ALLOC - refinement;
uint increase = min(wish,remainder/num_stream_channels);
_first_refinement_alloc[band] += increase;
remainder -= increase*_num_stream_channels;
}
// Carryover
carryover = remainder;
}
 	
// All bits for uncoded bands go to lognorm refinement.
for (uint band = _num_coded_bands; band < _num_effective_bands; band++) {
 	
// _alloc[band] evenly distributed
// (alloc_band] % remainder == 0 is guaranteed)
_first_refinement_alloc[band] = _alloc[band] / _num_stream_channels;
_alloc[band] = 0;
}
 	
// return
return carryover;
}
	9.9.13.17.1

9.9.13.17 Function Used by compute_first_lognorm_refinement_alloc

9.9.13.17.1 adjust_refinement

The function *adjust_refinements()* adjusts initial refinements.

Table 9-71: adjust_refinements

Syntax	Reference

// the function adjust_refinements adjusts initial refinements.
//
// parameter[in] tentative: initial estimate for refinement
// parameter[in] num_bins: number of bins in band
// parameter[in] band: band index.
//return: adjusted refinement.
 	
LNFP adjust_refinements(LNFP tentative, uint num_bins, uint band) {
 	
LNFP adjusted = tentative;
 	
if (tentative < 3 * num_bins) {
int boost_multiplier = 1;	

Syntax	Reference

`    if (tentative < 2* num_bins) {`
`      boost_multiplier = 2;`
`    }`
 	
`    LNFP boost = LOGNORM_REFINEMENT_BOOST_TABLE[band]*boost_multiplier;`
 	
`    adjusted += boost*boost_multiplier;`
`  }`
 	
`  return adjusted;`
`}`
	10.4.5

9.9.14 do_first_lognorm_refinement

The function *do_first_lognorm_refinement()* shown in Table 9-72 applies a first lognorm refinement, using refinement values read from the bitstream, and controlled by *_refinement_alloc*.

Table 9-72: do_first_lognorm_refinement

Syntax	Reference

`// apply a first lognorm refinement`
`//`
`// parameter[out] refinement_bits: refinement bit values as read from bitstream.`
 	
`void do_first_lognorm_refinement() {`
 	
`  uint refinement_bits[][];`
 	
`  // read refinement values from bitstream`
`  for (uint band=0;band < _num_effective_bands;band++) {`
 	
`    uint nbits = _first_refinement_alloc[band];`
 	
`    for (uint channel = _first_effective_channel;`
`         channel < _last_effective_channel;`
`         channel++) {`
`      refinement_bits[channel[band]= ReadUint(nbits);`
`    }`
`  }`
 	
`  // apply refinements`
`  do_lognorm_refinement(_first_refinement_alloc, refinement_bits,NULL);`
`}`
	8.2.8
	
	9.9.15.1

9.9.15 Functions Supporting do_first_lognorm_refinement

9.9.15.1 do_lognorm_refinement

The function *do_lognorm_refinement()* applies a lognorm refinement, and increases the precision of band energies. This function is called twice: the first time after completion of bit allocation computations (first refinement, clause 9.9.14), the second time after completion of band decoding (late refinement, clause 9.9.15.3).

Table 9-73: do_lognorm_refinement

Syntax	Reference

`// apply a lognorm refinement`
`//`
`// parameter[in] refinement_alloc: the number refinement bits per band`
`// parameter[in] refinement_bits: the refinement value per channel per band`
`// parameter[in] refinement_level: the refinement levels`
 	
`void do_lognorm_refinement(uint refinement_alloc[],`
`                           uint refinement_bits[][],`
`                           uint refinement_level[]) {`
 	
`  for (uint band = 0; band < _num_effective_bands; band++) {`	

Syntax	Reference

`    uint num_bits = refinement_alloc[band];`
`    uint level = refinement_level ? refinement_level[band] : 0;`
 	
`    for (uint channel = _first_effective_channel;`
`                         channel < _last_effective_channel;`
`                         channel++) {`
 	
`      if (num_bits > 0) {`
 	
`        bits = refinement_bits[channel][band];`
`        LNFP correction = compute_refinement_correction(num_bits, bits, level);`
`        _lognorm_with_offset[channel][band] += correction;`
`      }`
`    }`
`  }`
 	
`  // remove offsets from lognorm`
`  transfer_lognorms();`
`}`
	9.9.15.2
	
	9.9.15.3

9.9.15.2 compute_refinement_correction

The function *compute_refinement_correction()* computes the refinement correction given a number of bits and a refinement level.

Table 9-74: compute_refinement_correction

Syntax

`// compute a refinement correction`
`//`
`// parameter[in] nbit: number of bits`
`// parameter[in] bits: bit values`
`// parameter[in] level: level of correction`
`// return: refinement value`
 	
`LNFP compute_refinement_correction(uint nbits, uint bits, uint level) {`
 	
`  LNFP half_level = 1.0 / (1 << (level+1));`
`  LNFP refinement = LNFP(2*bits+1) / (1 << (nbits+level+1));`
`  refinement -= half_level;`
 	
`  return refinement;`
`}`

9.9.15.3 transfer_lognorms

The function *transfer_lognorms()* copies the matrix *_lognorm_with_offset* to the matrix *_lognorm*, removing the offsets introduced at encoding time.

Table 9-75: transfer_lognorms

Syntax

`// remove lognorm offsets.`
`//`
 	
`transfer_lognorms() {`
 	
`  for (uint channel=_first_effective_channel;`
`       channel <= _last_effective_channel;`
`       channel++) {`
 	
`    for (uint band=0; band<_num_effective_bands;band++) {`
 	
`      LNFP offset = _config->lognorm_offset[band];`
`      if (offset == LOGNORM_MINUS_INFINITY) {`
`        _lognorm[channel][band = LOGNORM_MINUS_INFINITY;`
`      }`
`      else {`
`        _lognorm[channel][band] = _lognorm_with_offset[channel][band] + offset;`

Syntax
} } } }

9.9.16 do_late_lognorm_refinement

The function *do_late_lognorm_refinement()*, shown in Table 9-76 applies a late lognorm refinement, using refinement values read from the bitstream, and controlled by *_late_refinement_alloc*. This late refinement is applied after processing of all bands.

Table 9-76: do_late_lognorm_refinement

Syntax	Reference

// apply a late lognorm refinement
//
// parameter[out] refinement_bits: refinement bit values as read from bitstream.
 	
void do_late_lognorm_refinement() {
  compute_late_lognorm_refinement_alloc();
 	
  uint late_refinement_bits[][];
 	
  // read refinement values from bitstream
  for (uint band=0;band < _num_effective_bands;band++) {
 	
  uint nbits = _late_refinement_alloc[band];
 	
  for (uint channel = _first_effective_channel;
  channel < _last_effective_channel;
  channel++) {
 	
  late_refinement_bits[channel[band]= ReadUint(nbits);
  }
  }
 	
  // aply refinements
  do_lognorm_refinement(_late_refinement_alloc,
late_refinement_bits,_first_refinement_alloc);
}
	9.9.17.1
	
	9.9.15.1

9.9.17 Function Supporting do_late_lognorm_refinement

9.9.17.1 compute_late_lognorm_refinement_alloc

The function *compute_late_lognorm_refinement_alloc()*, shown in Table 9-77, computes the bit allocations for a final lognorm refinement. This function uses data from the first refinement allocations, and from bits possibly saved for late refinement during band reconstruction.

Table 9-77: compute_late_lognorm_refinement_alloc

Syntax	Reference

// compute late lognorm refinement allocations
 	
void compute_late_lognorm_refinement_alloc() {
 	
  const uint MAX_REFINEMENT_ALLOC = 8;
 	
  uint remaining_bits = BitsLeft();
  remaining_bits = _num_effective_channels*(remaining_bits/_num_effective_channels);
 	
  set(_late_refinement_alloc,0);
 	
  // transfer late refinement bits saved during band reconstruction
  for (uint band=0; band < _num_effective_bands; band++) {
  if (_first_refinement_alloc[band] < MAX_REFINEMENT_ALLOC) {
  uint bits = _saved_late_refinement_alloc[band];
	8.2.4

Syntax	Reference

`      _late_refinement_alloc[band] = bits;`
`      remaining_bits -= bits * _num_effective_channels; // bits are no longer available`
`    }`
`  }`
 	
`  uint prev_remaining_bits = 0;`
`  uint rounds = 0;`
`  while (remaining_bits && (rounds < 2 || prev_remaining_bits != remaining_bits) ) {`
 	
`    prev_remaining_bits = remaining_bits;`
`    for ( uint band=0; (band < _num_effective_bands) && remaining_bits; band++ ) {`
 	
`      int nbits = _first_refinement_alloc[band] + _late_refinement_alloc[band];`
`      if ( nbits < MAX_REFINEMENT_ALLOC &&`
`                    (rounds>0 || _saved_late_refinement_alloc[band]==0) ) {`
`        // skip if band already got the max possible refinement bits or, in the`
`        // first round, if the band got "saved" late refinements during reconstruction`
`        _late_refinement_alloc[band]++;`
`        remaining_bits -= num_eff_channels;`
`      }`
`      rounds ++;`
`    } // for`
`  } // while`
`}`	

9.9.18 apply_lognorms

The function *apply_lognorms()* in Table 9-78 restores band spectral power values for all channels and all bands.

Table 9-78: apply_lognorms

Syntax

`// restore spectral band energies`
`//`
`apply_lognorms() {`
 	
`  for ( uint channel=0; channel < _num__stream_channels; channel++ ) {`
`    for ( uint band=0; band < _num_effective_bands; band++ ) {`
 	
`      float band_norm;`
`      LNFP lognorm = _lognorm[channel][band];`
`      if (lognorm == LOGNORM_MINUS_INFINITY) {`
`        band_norm = 0;`
`      else {`
`        band_norm = pow(2.0,(float)lognorm);`
`      }`
 	
`      uint band_start = _config->band_boundaries[band];`
`      uint band_end = _config->band_boundaries[band+1];`
`      for (index = band_start; index < band_end; band++) {`
 	
`        _stream_buffers[channel][band] *= band_norm;`
`      }`
`    }`
`  }`
`}`

9.10 Decoding Algorithms

9.10.1 Band Reconstruction

The functions in this clause reconstruct the frequency domain representation of ACE streams. The top level function *read_all bands()* sequentially reads all the coded band for a given stream from the bitstream, adjusting the number of bits available for reading each band depending on the number of bits that have been spent on previous bands. For stereo bands, the method of stereo decoding (*leftright* or *midside*) is established prior to reading and reconstructing from bitstream data. At the lowest level, band data are retrieved from the bitstream using the function *read_one_band_mono()*, reading a single band for a single channel from the bitstream.

9.10.2 Functions to Perform Band Reconstruction

9.10.2.1 Band Reconstruction Constants and Definitions

Table 9-79 contains a list of constants and definitions is shared by all band reconstruction modules.

Table 9-79: Band Reconstruction Constants and Definitions

Syntax	Reference

`const uint UNALLOC_UNIFORM_THRESHOLD = 4;  // Threshold for fractional`
`// forwarding`
`const RBFP FRAC_BALANCE_FOR_NEXT_BAND = RBFP::Fraction(7,5); // 7/32: Fraction for`
`// forwarding unspent bits`
	8.4.1
	8.4.2

9.10.2.2 read_all_bands

The function *read_all_bands()*, shown in Table 9-80, reads the frequency domain data for a mono (one channel) or stereo (two channels) stream. Ultimately this function resolves to multiple calls to *read_one_band_mono()*.

Table 9-80: read_all_bands

Syntax	Reference
`// reconstruct band data from the bitstream.`	
`uint read_all_bands () {`	
`  uint unallocated_bits = _unallocated_bits;`	
`  set(_saved_late_refinement_alloc, 0); // clear late refinements`	
`  uint remaining_bits = BitsLeft();`	8.2.4
`  float mdct_channel[][] = _stream_buffers;`	
`  uint num_before_bits = BitsUsed();`	8.2.3
`  // As band are processed, the variable 'running_balance' will measure the`	
`  // difference between the bits originally allocated to the bands`	
`  // processed so far, and the bits actually spent on those bands.`	
`  // A positive balance means less bits were spent that were originally`	
`  // allocated; the positive balance becomes available to the bands that remain`	
`  // to process. Conversely, a negative balance means that more bits were`	
`  // spent than originally allocated; bands that remain to process will have`	
`  // less bits available than originally.`	
`  // The initial value of running_balance is the number of bits from the original`	
`  // bit budget for the stream that were not specifically allocated to any band.`	
`  int running_balance = unallocated_bits;`	
`  int total_late_refinement_saved = 0; // total bits saved for late refinement`	
`  for (uint band=0;band<num_effective_bands;band++) {`	
`    uint bits_used = BitsUsed();`	8.2.3
`    remaining_bits = _total_bits - bits_used - total_late_refinement_saved;`	
`    uint allocated_bits = 0;`	
`    if (band < num_coded_bands) {`	

Syntax	Reference

`// Get the part of running_balance that will be allocated to this band,`
`// depending on distance to the last coded band. Notice that this`
`// quantity may be negative, if the running balance is negative.`	

`int running_balance_current =` `distribute_running_balance(running_balance, _num_coded_bands - band);`	9.10.2.3.1

`// Determine the actual allocation to the current band by adding`
`// the part of the running balance computed above to the original`
`// bit allocation. However, the allocation may never exceed`
`// the remaining bit budget for the stream as given in remaining_bits.`	

`allocated_bits = min(remaining_bits, _alloc[band] + running_balance_current);`
`allocated_bits = max(0, allocated_bits);`
`}`	

`uint stereo_coding_type = read_stereo_coding_type(band);`	9.10.3.3

`uint local_bits_spent = 0;`
`switch(stereo_coding_type) {`
`case kBandStereoCodingLeftRight: {`
`// read a leftright encoded band`
`local_bits_spent = read_one_band_leftright(band,`
`allocated_bits,`
`remaining_bits,`
`mdct_channel)`
`break;`
`}`
	9.10.3.4

`case kBandStereoCodingMidSide:`
`case kBandStereoCodingMidSideMono: {`
`// read a midside encoded band (mono or stereo)`
`local_bits_spent = read_one_band_midside(band,`
`allocated_bits,`
`remaining_bits,`
`mdct_channel);`
`break;`
`}`
	9.10.3.6

`case kBandStereoCodingMono: {`
`// read a mono band`
`local_bits_spent = read_one_band_mono(band,`
`allocated_bits,`
`remaining_bits,`
`mdct_channel[0]);`
`break;`
`}`
	9.10.2.3.2

`default: {`
`abort("invalid stereo_coding_type");`
`}`
`} // switch(stereo_coding_type)`	

`// bits consumed by this band` `int bits_used = BitsUsed() - bits_used;`	8.2.3

`// band surplus (or deficit if negative)`
`int band_surplus = alloc[band] - bits_used; // difference from budget`
`// update running balance`
`running_balance += band_surplus;`
`int bits_unused = allocated_bits - bits_used;`
`int ref_bits = save for_late_refinement(bits_unused, running_balance, band);`
	9.10.2.3.3

`// returned ref_bits is per channel, multiply by number of channels`
`running_balance -= ref_bits * _num_effective_channels;`
`total_late_refinement_saved += ref_bits * _num_effective_channels;`
`} // for (uint band=0;band<num_effective_bands;band++)`
`return BitsUsed() - num_before_bits;`
`}`
	8.2.3

9.10.2.3 Functions Supporting read_all_bands

9.10.2.3.1 distribute_running_balance

The function *distribute_running_balance()* redistributes bits that have not been spent on the current band to future bands.

Table 9-81: distribute_running_balance

Syntax

```
// Redistribute unspent bits to future bands.
//
// parameter[in] running_balance: current surplus / deficit
// parameter[in] dist_to_end: distance to last band
// return: updated running_balance

int distribute_running_balance(int running_balance, uint dist_to_end) {

  if ( dist_to_end > UNALLOC_UNIFORM_THRESHOLD) {
      int sign = 1;
      if ( running_balance < 0 ) {
          sign = -1;
          running_balance = -running_balance;
      }
      RBFP curr_balance_fp = FRAC_BALANCE_FOR_NEXT_BAND*running_balance;

      int curr_balance = (int) curr_balance_fp;
      if ( sign < 0 )
          curr_balance = -curr_balance;
      return curr_balance;
  }
  else  {
      // divide balance by number of bands remaining to process
      return running_balance / (int) dist_to_end);
  }
}
```

9.10.2.3.2 read_one_band_mono

The function *read_one_band_mono()*, in Table 9-82, reads a single band of MDCT coefficients from an ACE bitstream.

Table 9-82: read_one_band_mono

Syntax	Reference

`uint read_one_band_mono(uint band,`
`  uint allocated_bits, uint remaining_bits,`
`  float vector[]) {`
 	
`  uint band_start = _config->band_boundaries[band];`
`  uint band_end = _config->band_boundaries[band+1];`
`  uint band_size = _config->bandsize[band];`
 	
`  uint local_bits_spent = 0;`
 	
`  uint current_depth = 0;`
`  vector = vector + band_start;`
 	
`  local_bits_spent += read_split_vector(allocated_bits,`
`  remaining_bits,`
`  band_size,`
`  _num_band_blocks,`
`  current_depth,`
`  1.0,`
`  vector);`
`  reshape_band(_frame_duration,`
`  band_start, band_end,`
`  _num_band_blocks, _num_mdct_blocks,`
`  vector);`
 	
`  return local_bits_spent;`
`}`
	9.10.4.3
	
	9.10.5.4

9.10.2.3.3 save_for_late_refinement

The function *save_for_late_refinement()* saves, if conditions are met, bits that were allocated to but not used by the current band for late refinement of the band lognorm.

Table 9-83: save_for_late_refinement

Syntax

```
// Save unused bits for late refinement
//
// parameter[in] bits_unused: bits allocated to but not used by current band
// parameter[in] running_balance: current surplus / deficit
// parameter[in] band: index of current band
// return: number of bits saved, per channel, for late refinement
//         of current band lognorm

int save_for_late_refinement(int bits_unused, int running_balance, uint band) {

  int bits = 0;
  if ( _is_highres && bits_unused>0 && running_balance >= _num_effective_channels )
  {
    bits = 1;
  }
  _saved_late_refinement_alloc[band] = bits;
  return bits;
}
```

9.10.3 Stereo

9.10.3.1 Stereo Overview

The functions in this clause handle the stereo aspects of ACE stream. The function *read_stereo_coding_type()* establishes the stereo coding type for a given band, whereas the functions *read_one_band_leftright()* and *read_one_band_midside()* read and reconstruct a leftright stereo representation from a leftright or midside encoded frequency band. Auxiliary functions are used to determine the bit allocations assigned to the stereo elements in the bitstream.

9.10.3.2 Stereo Constants and Definitions

Table 9-84 lists the constants and definitions is shared by all stereo modules.

Table 9-84: Stereo Constants and Definitions

Syntax

```
const uint kBandStereoCodingMono = 0;          // Mono, single channel
const uint kBandStereoCodingMidSideMono = 1;  // Stereo, single channel encoded
const uint kBandStereoCodingLeftRight = 2;    // Stereo, leftright representation
const uint kBandStereoCodingMidSide = 3;      // Stereo, midside representation

const uint kLrStereoAllocation5050 = 0;        // 50/50 bit distribution
const uint kLrStereoAllocation6040 = 1;        // 60/40 bit distribution
const uint kLrStereoAllocation4060 = 2;        // 40/60 bit distribution
const uint kLrStereoAllocation7525 = 3;        // 75/25 bit distribution
const uint kLrStereoAllocation2575 = 4;        // 25/75 bit distribution

const uint MIN_DIFFERENCE_FOR_REBALANCING = 3; // Threshold for rebalancing

const uint LR_ALLOC_NUM_MODES = 5;             // Number of allocation modes
const uint LR_ALLOC_NUM_PARAMS = 3;            // Number of allocation params
const uint LR_ALLOC_NUM_REGIONS = 3;           // Number of allocation regions

const uint LOW_REGION_THRESHOLD = 4;           // Number of bands in first region
const uint MID_REGION_THRESHOLD = 12;          // Number of bands in first and
                                               // second region.


const uint LR_ALLOC_PARAMS[LR_ALLOC_NUM_MODES][LR_ALLOC_NUM_PARAMS] =
{
  // These parameters are used to calculate the percentage L/R allocations as:
```

Syntax

// (table[*][0] * bit_budget + table[*][2]) / table[*][1]
{ 1u, 2u, 1u }, // 50/50 split mode
{ 6u, 10u, 5u }, // 60/40 split mode
{ 6u, 10u, 5u }, // 40/60 split mode
{ 3u, 4u, 2u }, // 75/25 split mode
{ 3u, 4u, 2u } // 25/75 split mode
};

9.10.3.3 read_stereo_coding_type

read_stereo_coding_type(), shown in Table 9-85, establishes the stereo coding type for the current band.

Table 9-85: read_stereo_coding_type

Syntax

// Establishes the stereo coding type for the current band.
//
// parameter[in] band: current band index.
// return: stereo coding type of the current band
 	
uint read_stereo_coding_type(uint band) {
 	
if (!is_stereo) {
stereo_coding_type = kBandStereoCodingMono;
}
else {
stereo_coding_type = _stereo_flag[band];
}
 	
return stereo_coding_type;
}

9.10.3.4 read_one_band_leftright

The function *read_one_band_leftright()*, shown in Table 9-86, reads a frequency band of a leftright encoded stereo stream. The actual bitstream reading is handled by the function *read_one_band_mono()*.

Table 9-86: read_one_band_leftright

Syntax	Reference

// Read one band of a leftright encoded stereo stream.
//
// parameter[in] band: index of frequency band
// parameter[in] allocated_bits: allocated bits for reading stereo band
// parameter[in] remaining_bits: remaining bits for reading stereo stream
// parameter[out] vector: reconstructed stereo band (2 coefficient arrays)
// return: the number of bits spent in reconstructing the stereo band
 	
uint read_one_band_left_right(uint band,
uint allocated_bits,
uint remaining_bits,
float vector[][]) {
 	
uint alloc_channel0;
uint alloc_channel1;
 	
uint local_bits_spent = 0;
 	
// compute bit allocations
compute_leftright_allocations(band,
allocated_bits,
alloc_channel0,
alloc_channel1);
 	
// read left channel
local_bits_spent = read_one_band_mono(band,
alloc_channel0,
remaining_bits,
vector[0]);
	9.10.3.5.1
	
	9.10.2.3.2

Syntax	Reference

`  // read right channel`
`  remaining_bits -= local_bits_spent;`
`  local_bits_spent = read_one_band_mono(band,`
`  alloc_channel1,`
`  remaining_bits,`
`  vector[1]);`
 	
`  return local_bits_spent;`
`}`
	9.10.2.3.2

9.10.3.5 Functions Called from read_one_band_left_right

9.10.3.5.1 compute_leftright_allocations

The function *compute_leftright_allocations()* distributes the bits available for a leftright encoded stereo stream over the left and right channel. When the left or the right channel is signaled to be silent, all bits will be allocated to the remaining channel.

Table 9-87: compute_leftright_allocations

Syntax	Reference

`// Distribute allocated bits over left and right channel.`
`//`
`// parameter[in] band: index of band`
`// parameter[in] allocated_bits: bit allocation for band`
`// parameter[out] bits_l: bits allocated to left channel`
`// parameter[out] bits_r: bits allocated to right channel.`
 	
`void compute_leftright_allocations(uint band,`
`  uint allocated_bits,`
`  uint &bits_l,`
`  uint &bits_r) {`
 	
`  // handle silent right channel`
`  if (coding_mode == kCodingLeft) {`
`  bits_l = allocated_bits;`
`  bits_r = 0;`
`  return;`
`  }`
 	
`  // handle silent left channel`
`  if (coding_mode == kCodingRight) {`
`  bits_l = 0;`
`  bits_r = allocated_bits;`
`  return;`
 	
`  // compute the mode for the band`
`  uint mode = _lr_alloc_region_mode[compute_region(band)];`
	9.10.3.5.2

`  // compute the formula parameters`
`  uint factor = LR_ALLOC_PARAMS[mode][0];`
`  uint denominator = LR_ALLOC_PARAMS[mode][1];`
`  uint rounding_offset = LR_ALLOC_RATO_PARAMS[mode][2];`
 	
`  // Determine whether L or R should get the larger percentage of bits`
`  bool larger_percentage_for_l;`
`  if (mode == kLrStereoAllocation5050) {`
`  if (lognorm[0][band] >= lognorm[1][band])`
`  larger_percentage_for_l = true;`
`  else`
`  larger_percentage_for_l = false;`
`  }`
`  else if (mode == kLrStereoAllocation6040 ||`
`  mode == kLrStereoAllocation7525) {`
`  larger_percentage_for_l = true;`
`  }`
`  else {`
`  larger_percentage_for_l = false;`
`  }`
 	
`  // Allocate bits`
`  if (larger_percentage_for_l) {`	

Syntax	Reference

bits_l = (factor * allocated_bits + rounding_offset) / denominator;
bits_2 = allocated_bits - bits_l;
} else {
bits_r = (factor * allocated_bits + rounding_offset) / denominator;
bits_l = allocated_bits - bits_r;
}
}	

9.10.3.5.2 compute_region

The function *compute_region()* computes the leftright allocation region associated with a frequency band.

Table 9-88: compute_region

Syntax

// Compute region associated with a band.
//
// parameter[in] band: frequency band.
// return: region associated with band (in {0,1,2})
 	
uint compute_region(uint band) {
 	
if (band < LOW_REGION_THRESHOLD) {
return 0;
}
else if (band < MID_REGION_THRESHOLD) {
return 1;
}
else {
return 2;
}
}

9.10.3.6 read_one_band_midside

The function *read_one_band_midside()* reads a frequency band of a midside encoded stereo stream. The actual bitstream reading is handled by the function *read_one_band_mono()*.

Table 9-89: read_one_band_midside

Syntax	Reference

// Read one band of a midside encoded stereo stream.
//
// parameter[in] band: index of frequency band
// parameter[in] allocated_bits: allocated bits for reading stereo band
// parameter[in] remaining_bits: remaining bits for reading stereo stream
// parameter[in] int_beta_array: bit allocation partitioning data (beta)
// parameter[out] vector: reconstructed stereo band (2 coefficient arrays)
// return: the number of bits spent in reconstructing the stereo band
 	
uint read_one_band_midside(uint band,
uint allocated_bits,
uint remaining_bits,
int int_beta_array,
float vector[2][]) {
 	
uint N = _config->bandsize(band);
LNFP logN = _config->log_band_size[band];
 	
float norm_m, norm_s;
uint bits_m, bits_s;
bool is_final;
 	
allocated_bits = min(allocated_bits,remaining_bits);
 	
uint beta_bits = compute_split_parameters(allocated_bits,
remaining_bits,
N, N, logN+1,
bits_m, bits_s,
norm_m, norm_s,
	9.10.4.4.2

Syntax	Reference

is_final,
int_beta_array + band);
 	
uint local_bits_spent = beta_bits;
 	
if (bits_m >= bits_s) {
 	
// Read mid channel
local_bits_spent += read_one_band_mono(band, bits_m,remaining_bits, vector[0]);
	9.10.2.3.2

// Rebalance allocation
uint bit_difference = bits_m - local_bits_spent;
if ((bit_difference > MIN_BIT_DIFFERENCE_FOR_REBALANCING) && (!is_final))) {
bits_s += bit_difference - MIN_BIT_DIFFERENCE_FOR_REBALANCING;
}
 	
// read side channel
local_bits_spent += read_one_band_mono(band, bits_s,remaining_bits,vector[1]);
	9.10.2.3.2

}
else {
 	
// Read side channel
local_bits_spent += read_one_band_mono(band,bits_s,remaining_bits,vector[1]);
	9.10.2.3.2

// Rebalance allocation
uint bit_difference = bits_s - local_bits_spent;
if ((bit_difference > MIN_BIT_DIFFERENCE_FOR_REBALANCING) && (!is_final))) {
bits_m += bit_difference - MIN_BIT_DIFFERENCE_FOR_REBALANCING;
}
 	
// Read mid channel
local_bits_spent += read_one_band_mono(band,bits_m,remaining_bits,vector[0]);
	9.10.2.3.2

}
 	
// Transform to leftright
midside_to_leftright(band,norm_m, norm_s, vector);
	9.10.3.7

// Return number of bits spent
return local_bits_spent;
}	

9.10.3.7 midside_to_leftright

The function *midside_to_leftright()* transforms the midside representation of a stereo signal to a leftright representation of a stereo channel.

Table 9-90: midside_to_leftright

Syntax	Reference

// Transform midside representation to leftright representation.
//
// parameter[in] band: the index of the frequency band
// parameter[in] norm_m: the L2-norm of the mid-channel
// parameter[in] norm_s: the L2-norm of the side-channel
// parameter[in/out] vector: the pair of coefficient arrays to be transformed.
 	
void midside_to_leftright(uint band,
float norm_m,
float norm_s,
float vector[2][]) {
 	
uint band_start = _config->band_boundaries[band];
uint band_end = _config->band_boundaries[band+1];
uint band_size = _config->band_size[band];
 	
for (int n=band_start;n<band_end;n++) {
float m = norm_m*vector[0][n];
float s = norm_s*vector[1][n];
vector[0][m] = m;
vector[1][m] = s;
}	

Syntax	Reference

normalize(band_size,1.0,vector[0]+band_start);
normalize(band_size,1.0,vector[1]+band_start);
}
	9.10.4.9.7
	9.10.4.9.7

9.10.4 Vector Quantization (VQ)

9.10.4.1 Overview of VQ

The functions in this clause and sub-clauses specify how frequency domain coefficients are read from the bitstream. At the lowest level the function *read_nonsplit_vector()* reads a segment of data from the bitstream given an index for an unsigned pyramid code, and a sequence of sign bits. At the top-level the function *read_split_vector()* recursively splits, until is able to call the function *read_nonsplit_vector()*.

9.10.4.2 VQ Constants and Definitions

The list of constants and definitions shown in Table 9-91 is shared by all vector quantization modules.

Table 9-91: VQ Constants and Definitions

Syntax

const uint MIN_BIT_DIFFERENCE_FOR_REBALANCING = 3; // Threshold for rebalancing
const uint LOG_MAXIMUM_QUANTIZATION_LEVEL = 8; // Log of maximum quantization
// level
const uint MAXIMUM_QUANTIZATION_LEVEL = 256; // Maximum quantization level
// for partitioning parameter
// of recursive split
const uint MAX_DEPTH_LOWRES = 3; // Max depth of recursion in
// normal resolution. The max
// depth is increased by one
// when _is_highres is true.

9.10.4.3 read_split_vector

The function *read_split_vector()*, shown in Table 9-92, recursively reads a band of MDCT coefficients from the bitstream. The input parameter *gain* specifies the gain to be applied after reconstruction. This parameter is set to 1 at the top level. The depth parameter *depth* is to be set to 0 at the top level. The value of *num_blocks* is to be provided as *num_band_blocks*, as determined by *read_num_band_blocks*.

Table 9-92: read_split_vector

Syntax	Reference

// Recursively read a set of band coefficients from the bitstream
//
// parameter[in] allocated_bits: the number of bits allocated to this vector
// parameter[in] remaining_bits: the total number of available bits for this stream
// parameter[in] length: the length of this vector
// parameter[in] num_blocks: the number of time slots in this vector
// parameter[in] current_depth: the depth of the recursion
// parameter[in] gain: the gain to be applied to the reconstructed vector
// parameter[out] vector: the reconstructed vector
 	
void read_split_vector(uint allocated_bits,
uint remaining_bits,
uint N,
uint num_blocks,
uint current_depth,
float gain,
float vector[]) {
 	
uint local_bits_spent = 0;
int allocated_bits = min(allocated_bits, remaining_bits);
if (is_split_required(allocated_bits, N, current_depth)) {
 	
// Split the vector in two halves
	9.10.4.4.1

Syntax	Reference

int partition_depth = current_depth + 1;
uint N_1 = (N + 1) >> 1;
uint N_2 = N - N_1;
float vector_1[] = vector;
float vector_2[] = vector + N_1;
int partition_num_blocks = (num_blocks + 1) >> 1;
float norm_1, norm_2;
 	
LBFP logN = _config->log_elementary_band_size - current_depth;
bool is_final;
uint beta_bits = compute_split_parameters(allocated_bits,
	9.10.4.4.2

remaining_bits,
N, logN,
norm_1, norm_2,
bits_1, bits_2,
is_final);
int remaining_bits -= beta_bits;
allocated_bits -= beta_bits;
 	
// Recursive calls.
// Decode the half with most allocated bits first.
// If some of the allocated bits are not used,
// reassign them to the second half.
if (bits_1 >= bits_2) {
local_bits_spent = read_split_vector(bits_1,
	9.10.4.3

remaining_bits,
N_1,
partition_num_blocks,
partition_depth,
gain * norm_1,
vector_1);
 	
remaining_bits -= local_bits_spent;
uint bit_difference = bits_1 - local_bits_spent;
if ((bit_difference > MIN_BIT_DIFFERENCE_FOR_REBALANCING) && (!is_final))) {
bits_2 += bit_difference - MIN_BIT_DIFFERENCE_FOR_REBALANCING;
}
uint aux_bits_spent;
aux_bits_spent = read_split_vector(bits_2,
	9.10.4.3

remaining_bits,
N_2,
partition_num_blocks,
partition_depth,
gain * norm_2,
vector_2);
local_bits_spent += aux_bits_spent;
}
else {
local_bits_spent = read_split_vector(bits_2,
	9.10.4.3

remaining_bits,
N_2,
partition_num_blocks,
partition_depth,
gain * normalized_norm_2,
vector_2);
remaining_bits -= local_bits_spent;
uint bit_difference = bits_2 - local_bits_spent;
if ((bit_difference > MIN_BIT_DIFFERENCE_FOR_REBALANCING) && (!is_final)) {
bits_1 += bit_difference - MIN_BIT_DIFFERENCE_FOR_REBALANCING;
}
uint aux_bits_spent;
aux_bits_spent = read_split_vector(bits_1,
	9.10.4.3

remaining_bits,
N_1,
partition_num_blocks,
partition_depth,
gain * normalized_norm_1,
vector_1);
local_bits_spent += aux_bits_spent;
}
local_bits_spent += beta_bits;
}
// no split
else {
local_bits_spent = read_nonsplit_vector(allocated_bits, remaining_bits,
	9.10.4.5

N, num_blocks, gain, vector[]); }	

Syntax	Reference

// return
return local_bits_spent;
}	

9.10.4.4 Functions Supporting read_split_vector

9.10.4.4.1 is_split_required

The function *is_split_required()* indicates whether or not the segment being read needs to be read in parts.

Table 9-93: is_split_required

Syntax	Reference

// Determine whether or not a recursive split is required.
//
// parameter[in] allocated_bits: the bits available for reading band coefficients
// parameter[in] N: the length of the vector to be read
// parameter[in] depth: the recursive depth of the recursive split
// return: whether or not a split is required
 	
bool is_split_required(uint allocated_bits, uint N, uint depth) {
uint max_depth = MAX_DEPTH_LOWRES + _is_highres ? 1 : 0;
if ((depth > max_depth) || (N < 4)) {
return false;
}
else {
return bits > MAX_BITS_FOR_N[N-2];
}
}
	10.7.2

9.10.4.4.2 compute_split_parameters

The function *compute_split_parameters()* computes the bit assignments and norm values for vector that is read as two segments from a bitstream. Values are computed for the first and second segment. This function is also used by the stereo module, determining the bit allocations for left and right channels.

Table 9-94: compute_split_parameters

Syntax	Reference

// Compute bit allocations and norms for first and second segment in a split.
//
// parameter[in] allocated_bits: the number of bits allocated
// parameter[in] remaining_bits: the total number of bits available
// parameter[in] N: the length of the vector to be retrieved
// parameter[in] logN: a fixedpoint approximation to log2(N)
// parameter[out] bits1: number of bits to be assigned to left segment
// parameter[out] bits2: number of bits to be assigned to right segment
// parameter[out] norm1: norm for the left segment
// parameter[out] norm2: norm for the right segment
// parameter[out] is_final: true if one of the segments has zero norm.
 	
uint compute_split_parameters(uint allocated_bits,
uint remaining_bits,
uint N,
LBFP logN,
uint &bits1,
uint &bits2,
float &norm1,
float &norm2,
bool &is_final) {
 	
uint quantization_level = compute_quant_level(allocated_bits, N, logN);
uint local_bits_spent = BitsUsed();
uint index = ReadUniform(quantization_level+1);
local_bits_spent = BitsUsed() - local_bits_spent
allocated_bits -= local_bits_spent;
 	
int int_beta;
	9.10.4.4.3
	
	8.2.11
	8.2.3

Syntax	Reference

if (index <= quantization_level/2) {
  int_beta = BETA_DEQUANT_TABLE[quantization_level/2 - 1][index];
}
else {
  int_beta =
  -BETA_DEQUANT_TABLE[quantization_level/2 - 1][quantization_level/2-index];
}
if (index == 0) {
  norm1 = 0.0;
  norm2 = 1.0;
  bits1 = 0;
  bits2 = bits;
  if_final = true;
}
else if (index == quantization_level) {
  norm1 = 1;
  norm2 = 0;
  bits1 = bits;
  bits2 = 0;
  is_final = true;
}
// quantization_level guaranteed to be even
else {
 	
  // fixedpoint beta in (-1 ..+1)
  BQFP fp_beta = (BQFP(int_beta) / quantization_level) - 0.5;
 	
  // fixedpoint delta
  BQFP fp_delta = -(N-1)*fp_beta;
 	
  // beta and delta
  float beta = (float) fp_beta;
  int delta = Round(fp_delta);
 	
  // compute norm values
  float beta_term = pow(2.0,2.0*beta); // 2^(2*beta)
  norm1 = sqrt(beta_term / (1.0 + beta_term));
  norm2 = sqrt(1.0 / (1.0 + beta_term));
 	
  // compute bit allocations
  int bits1 = min(bits, (bits - delta + 1) / 2);
  if (bits1 < 0) bits1 = 0;
  int bits2 = bits - bits1;
 	
  // final flag
  is_final = false;
}
return local_bits_spent;
}
	10.7.5

9.10.4.4.3 compute_quant_level

The function *compute_quant_level()* computes the quantization level of the log ratio of the norms of the left and right half a partitioned vector as encoded in the bitstream. The computed quantization level depends on the number of bits available and the length of the vector to be partitioned.

Table 9-95: compute_quant_level

Syntax

// Compute partitioning
// parameter[in] bits: the number of bits available
// parameter[in] N: the length of the vector
// parameter[in] logN: an approximation to log2(N)
 	
uint compute_quant_level(uint bits, uint N, LBFP logN) {
 	
  LBFP fp_bits = bits;
  LBFP exponent_0 = (fp_bits + ((logN/2)-1) * (N-1)) / (N-1);
  LBFP exponent_1 = b - 2;
  LBFP exponent = (exponent_0 < exponent_1) ? exponent_0 : exponent_1;
 	
  if (exponent < 0.5) {

Syntax

`      return 1;`
`  }`
`  else if (exponent > LOG_MAXIMUM_QUANTIZATION_LEVEL) {`
`      return MAXIMUM_QUANTIZATION_LEVEL;`
`  }`
`  else {`
`      uint res = exp(exponent);`
`      return (res + 1) & even_mask;`
`  }`
`}`

9.10.4.5 read_nonsplit_vector

The function *read_nonsplit_vector()* reads a segment from of vector of a band of MDCT coefficients from the bitstream and sets the retrieved vector a specified gain value *gain*. The number of time slots *num_blocks* (corresponding to *num_band_blocks*) is provided to allow appropriate *reconditioning* of vectors encoded with very few pulses.

Table 9-96: read_nonsplit_vector

Syntax	Reference
`//`	
`//`	
`// parameter[in] N: length of vector`	
`// parameter[in] num_blocks: number of blocks in vector`	
`// parameter[in] gain: specified gain for output vector`	
`// parameter[out] vector: output vector`	
`// return: the number of bits read from the bitstream`	
`uint read_nonsplit_vector(uint allocated_bits, uint remaining_bits,` `                          uint N, uint num_blocks, float gain, float vector[]) {`	
`  uint local_bits_spent = BitsUsed();`	8.2.3
`  // Compute K value`	
`  uint K = get_best_k_for_n(allocated_bits, remaining_bits, N);`	9.10.4.9.4
`  // get alphabet size`	
`  uint alpha = pyramid(N,K);`	9.10.4.9.3
`  // retrieve index`	
`  uint index = ReadUniform(alpha);`	8.2.11
`  // construct absolute values`	
`  make_unsigned_pyramid_vector(index,N,K,vector);`	9.10.4.6
`  // retrieve signs`	

`  uint mask_length = hamming(N,vector);  // pyramid properties guarantee`
`                                    // mask_length <= 32`
	9.10.4.9.6

`  uint mask = 0;`	
`    for (int i=0;i<mask_length;i++) {`	
`    mask= (mask << 1 ) | ReadBit();`	8.2.3
`  }`	
`  // apply signs`	
`  for (int i=N-1;i>=0;i--) {`	
`    if (vector[i] != 0) {`	
`      if (mask&1) {`	
`        vector[i] *= -1;`	
`      }`	
`      mask >> 1;`	
`    }`	
`  }`	
`  // recondition`	
`  recondition(N,K,num_blocks,_reconditioning_level,vector);`	9.10.4.7.1
`  // spectral hole fill`	
`  do_spectral_hole_fill(N,K,vector);`	9.10.4.8
`  // normalize`	
`  normalize(N,gain,vector);`	9.10.4.9.7
`  // return the number of bits read`	

Syntax	Reference
`  return BitsUsed() - local_bits_spent;` `}`	8.2.3

9.10.4.6 make_unsigned_pyramid_vector

The function *make_unsigned_pyramid_vector()* reconstructs an element from an unsigned pyramid code with parameters N and K given an index value *index*.

Table 9-97: make_unsigned_pyramid_vector

Syntax	Reference

`// Reconstructs an element from an unsigned pyramid code`
`//`
`// parameter[in] index: the index value of the pulse vector`
`// parameter[in] N: the N parameter of the pyramid code`
`// parameter[in] K: the K value of the pyramid code`
`// parameter[out] vector: the reconstructed pulse vector.`
 	
`void make_unsigned_pyramid_vector(uint index, uint N, uint K, float vector[]) {`
 	
`  // assert index is valid`
`  assert(index < pyramid(N,K));`
	9.10.4.9.3

`  // recursively construct vector`
`  if (N == 1) {`
`    vector[0] = K;`
`    return;`
`  }`
`  else if (K == 0)) {`
`    set(vector, 0, N);`
`    return;`
`  }`
`  else {`
`    for(int j = K-1; j >=0; j--) {`
`      if (index >= pyramid(N,j)) {`
`        break;`
`      }`
`    }`
`    // set first value`
`    v[0] = K - j - 1;`
 	
`    // set parameters for recursion`
`    index -= pyramid(N,j);`
	9.10.4.9.3

`    // recurse`
`    make_unsigned_pyramid_vector(index,N-1,j+1,vector+1);`
`  }`
`}`	

9.10.4.7 Reconditioning

9.10.4.7.1 recondition

The function *recondition()* reconditions a raw pulse vector to mitigate for non-sparse original vectors that are being reconstructed with only a few pulses. Reconditioning is applied separately for each of the time slots (specified by *num_blocks*) in the vector. The input parameter *reconditioning_level* specifies to what degree reconditioning should be applied and is read from the bitstream.

Table 9-98: recondition

Syntax	Reference

`// parameter[in] N: vector length`
`// parameter[in] K: number of pulses`
`// parameter[in] num_blocks: number of blocks (time slots)`
`// parameter[in] reconditioning: recondition level (0..3)`
`// parameter[in/out] vector: vector that is being reconditioned`	

Syntax	Reference

void recondition(uint N,
uint K,
uint num_blocks,
uint reconditioning_level,
float vector[]) {
 	
const float RECONDITIONING_EXPO_FACTOR_1 = 3.0;
const float RECONDITIONING_EXPO_FACTOR_2 = 0.73;
 	
const uint kReconditioningLevelOff = 0;
const uint kReconditioningLevelLow = 1;
const uint kReconditioningLevelMedium = 2;
const uint kReconditioningLevelHigh = 3;
 	
const float RECONDITIONING_TABLE[] = {0, 0.36, 0.47, 0.75};
 	
if ((2*K >= num_blocks) ‖
(reconditioning_level == kReconditioningLevelOff) ‖ (K == 0)) {
return;
}
 	
float matrix[][];
 	
uint num_samples_block = N / num_blocks;
 	
float sigma = exp(RECONDITIONING_EXPO_FACTOR_1 *
pow(((float) K / (float) N),RECONDITIONING_EXPO_FACTOR_2)
RECONDITIONING_TABLE [reconditioning_level]);
 	
uint num_segments;
uint num_samples_segment;
 	
num_segments = num_samples_block / 4;
num_samples_segment = 4;
num_samples_left = num_samples_block % 4;
make_matrix_4(sigma,matrix);
	9.10.4.7.2

if (num_segments > 0) {
for (int blk=0;blk<num_blocks;blk++) {
 	
if (num_samples_left != 0) {
// Loop over the segments
for (int seg = 0; seg < num_segments; seg++) {
int index = blk * num_samples_block + seg * num_samples_segment;
apply_matrix(matrix,num_samples_segment,vector + index);
}
// flip to get sequence forward again
flip_vector(num_samples_block, vector + blk * num_samples_block);
}
	9.10.4.7.3
	9.10.4.7.4

// Loop over the segments
for (int seg = 0; seg < num_segments; seg++) {
int index = blk * num_samples_block + seg * num_samples_segment;
apply_matrix(matrix,num_samples_segment,vector + index);
}
 	
// de-interleave (in-place)
int index = blk * num_samples_block;
interleave_vector(num_samples_block, num_samples_segment, vector + index);
}
}
}
	9.10.4.7.3
	9.10.4.7.5

9.10.4.7.2 make_matrix_4

The function *make_matrix_4()* creates a reconditioning 4x4 matrix parameterized by a control *sigma*.

Table 9-99: make_matrix_4

Syntax
// parameter[in] sigma: reconditioning control parameter // parameter[out] matrix: reconditioning matrix

Syntax

```
void make_matrix_4(float sigma, float matrix[][]) {

  const float PI_4 = atan(1.0);
  const float PI_2 = acos(0.0);

  float a = cos(sigma * PI_4);
  float b = sin(sigma * PI_4);
  float c = sin(sigma * PI_2);

  float div = sqrt(a*a + 2*b*b + c*c);

  matrix[0][0] = +a / div;
  matrix[0][1] = -b / div;
  matrix[0][2] = +b / div;
  matrix[0][3] = -d / div;

  matrix[1][0] = +b / div;
  matrix[1][1] = +a / div;
  matrix[1][2] = +d / div;
  matrix[1][3] = +b / div;

  matrix[2][0] = -b / div;
  matrix[2][1] = -d / div;
  matrix[2][2] = +a / div;
  matrix[2][3] = +b / div;

  matrix[3][0] = +d / div;
  matrix[3][1] = -b / div;
  matrix[3][2] = -b / div;
  matrix[3][3] = +a / div;

}
```

9.10.4.7.3 apply_matrix

The function *apply_matrix()* applies a specified reconditioning matrix to a specified segment.

Table 9-100: apply_matrix

Syntax

```
// parameter[in] matrix: reconditioning matrix
// parameter[in] matrix_size: matrix dimension (size)
// parameter[in/out] vector: vector to which reconditioning is applied

void apply_matrix(float matrix[][], uint matrix_size, float vector[]) {

  float output[];

  for (int m = 0; m < matrix_size; m++) {
    output[m] = 0;
    for (int n = 0; n < matrix_size; n++) {
      output[m] += matrix[m][n] * vector[n];
    }
  }

  for (int m = 0; m < matrix_size; m++) {
    vector[m] = output[m];
  }
}
```

9.10.4.7.4 flip_vector

The function *flip_vector()* reverses the entries in a specified vector.

Table 9-101: flip_vector

Syntax

```
// parameter[in] length: length of the vector
// parameter[in/out] vector: vector to be flipped

void flip_vector(uint length, float vector[]) {

  for (int i = 0; i < length; i++) {
    float swap = vector[i];
    vector[i] = vector[length-i-1];
    vector[length-i-1] = swap;
  }
}
```

9.10.4.7.5 interleave_vector

The function *interleave_vector()* interleaves a vector modeled as matrix transpose.

Table 9-102: interleave_vector

Syntax

```
// parameter[in] length: length of the vector
// parameter[in/out] vector: vector to be interleaved.

void interleave_vector(uint length, uint size,float vector[]) {

  uint num_rows = size;
  uint num_cols = length / size;

  float output[];

  for (int m = 0; m < num_rows; m++) {
    for (int n = 0; n _num_cols; n++) {
      output[m*num_cols+n] = vector[n*num_rows+m];
    }
  }

  for (int m = 0; m < length; m++) {
    vector[m] = output[m];
  }
}
```

9.10.4.8 Spectral Hole Fill

9.10.4.8.1 Overview of Spectral Hole Fill

Spectral Hole Fill injects non-zero energy when no bits are available for vector quantization. The encoder indicates to the decoder using the *_shf_enabled* flag whether or not spectrall hole filling is appropriate. The method of Spectral Hole Filling is not normative. For the reader's convenience a simple method is described in the pseudo-code below, but actual implementation of an ACE decoder may choose different methods for energy injections.

9.10.4.8.2 do_spectral_hole_fill (informative)

The function *do_spectral_hole_fill(uint N, uint K, float vector[])* injects energy in an all-zero coefficient buffer. As this function is followed by normalization, the amount of energy injected is irrelevant, as long as it is non-zero.

Table 9-103: do_spectral_hole_fill

Syntax	Reference

`// Inject energy in coefficient buffer`
`//`
`// parameter[in] N: number of samples`
`// parameter[in] L: the K value of the pyramid code`
`// parameter[in/out] vector: the sample buffer`
 	
`void_do_spectral_hole_fill(uint N, uint K, float vector[]) {`
 	
`  // bail if any pulses`
`  if (K != 0) {`
`  return;`
`  }`
 	
`  for (int i=0;i<N;i++) {`
`  vector [i] = random_sign();`
`  }`
`}`
	9.10.7.4

9.10.4.9 Auxiliary functions

9.10.4.9.1 factorial

Table 9-104: factorial

Syntax	Reference

`// parameter[in] n: input value`
`// return: factorial(n) = n*(n-1)*...*1.`
 	
`uint factorial(uint n) {`
 	
`  if (n == 0) {`
`  return 1;`
`  }`
`  else {`
`  return n*factorial(n-1);`
`  }`
`}`	

9.10.4.9.2 choose

Table 9-105: choose

Syntax	Reference

`// parameter[in] n: input value`
`// parameter[in] k: input value`
`// return: the number of ways k elements can be chosen from a set of k elements`
 	
`uint choose(uint n, uint k) {`
 	
`  return  factorial(n) / ( factorial(k)* factorial(n-k));`
`}`	

9.10.4.9.3 pyramid

An unsigned pyramid code with parameters N and K is the set of vectors of length N with non-negative integer entries such that the sum of entries is equal to K. The function *pyramid()* computes the number of elements in an unsigned pyramid code.

Table 9-106: pyramid

Syntax	Reference

`// parameter[in] n:`
 	
`uint pyramid(uint N, uint K) {`
 	
`  return choose(N+K-1,K);`
`}`
	9.10.4.9.2

9.10.4.9.4 get_best_k_for_n

The function *get_best_k_for(uint N, uint allocated_bits)* computes the best value of *K* such that the signed pyramid *(N,K)* fits within the allocated budget.

Table 9-107: get_best_k_for_n

Syntax	Reference

`// compute the optimal value of K such that signed pyramid with parameters (N,K)`
`// satisfies the bit budget`
`//`
`// parameter[in] N: length of segment`
`// parameter[in] allocated_bits: number of bits allocated for segment retrieval`
`// return: the optimal value K`
 	
`int get_best_k_for_n(uint allocated_bits, uint remaining_bits, uint N) {`
 	
`  const uint MAXIMUM_K = 256;`
`  const uint MAXIMUM_K_LOWRES = MAXIMUM_K / 2;`
`  const uint NUM_INDEX_BITS = 32;`
`  const uint NUM_DIMENSIONS_WITH_MAX_K = 4;`
`  const uint NUM_DIMENSIONS_WITH_MAX_K_LOWRES = 5;`
`  const uint MAXIMUM_WORST_CASE_EXCESS = 8;`
`  const uint MINIMUM_BITS_FOR_LINEAR_EXCESS_BOUNDING = 6;`
`  const uint MAXIMUM_N_FOR_LINEAR_EXCESS_BOUNDING = 7;`
`  const uint A_COEFFICIENT_FOR_LINEAR_EXCESS_BOUNDING = 78;`
`  const uint B_COEFFICIENT_FOR_LINEAR_EXCESS_BOUNDING = 993;`
`  const uint FIXED_POINT_RESOLUTION = 9;`
`  const uint FIXED_POINT_ROUNDING = 1 << (FIXED_POINT_RESOLUTION - 1);`
 	
`  const uint MAX_BITS_FOR_N_LOWRES[] = {10, 17, 23, 29, 35};`
 	
`  typedef FixedPoint<FIXED_POINT_RESOLUTION> NKFP;`
 	
`  // do not spend more than available`
`  allocated_bits = min(allocated_bits, remaining_bits);`
 	
`  // stop when insufficient number of bits`
`  if (allocated_bits < 2) {`
`    return 0;`
`  }`
 	
`  // upperbound K`
`  uint maxK = max_k_for_n(N);`
	9.10.4.9.5

`  if (!_is_highres && (maxK > MAXIMUM_K_LOWRES)) {`
`      maxK = MAXIMUM_K_LOWRES;`
`  }`
 	
`  // upper bound number of bits`
`  int max_bits;`
`  if (!_is_highres && N < NUM_DIMENSIONS_WITH_MAX_K_LOWRES + 2) {`
`    max_bits = MAX_BITS_FOR_N_LOWRES[N-2];`
`  }`
`  else if (N < NUM_DIMENSIONS_WITH_MAX_K + 2) {`
`    max_bits = MAX_BITS_FOR_N[N-2];`
`  }`
`  else {`
`    max_bits = NUM_INDEX_BITS + min(N, maxK);`
`  }`
 	
`  allocated_bits = min(allocated_bits,max_bits);`
 	
`  // estimate K`
`  uint c1 = K_ESTIMATION_COEFFICIENTS[N-2][1];`
	10.7.2
	
	10.7.4

Syntax	Reference

```
NKFP fp_exponent(allocated_bits);
fp_exponent /= (N-1);

// Split integer and fractional parts of the exponent.
int int_exponent = intpart(fp_exponent);
fp_exponent.data(fracpart(fp_exponent));

// Calculate the term c1 * 2^{exp_frac}*2^fixed_point_resolution,
// where exp_frac is the fractional part of bits / (N - 1).
// Here the factor 2^fixed_point_resolution comes from the
// fixed point representation of c1.
NKFP fp_exp_factor = exp2(fp_exponent);
NKFP fp_exp_term = fp_exp_factor * c1;

// Multiply the number resulting from the computation above
// by 2^{exp_int - fixed_point_resolution},
// where exp_int is the integer part of bits / (N - 1),
// which was left appart,
// and 2^{-fixed_point_resolution} cancels the factor
// 2^fixed_point_resolution coming from the
// fixed point representation of c1 in tha computation above.
int exponent_difference = int_exponent - fixed_point_resolution;
if (exponent_difference >= 0)
  fp_exp_term <<= exponent_difference;
else
  fp_exp_term >>= -exponent_difference;

// Subtract the term c0 and round to nearest integer
NKFP fp_c0;
fp_c0.data(coeffs->c0);
NKFP fp_k = fp_exp_term - fp_c0;
int K = round(fp_k);

if (K < 1) {
  K = 1;
}
else if (K > maxK){
  K = maxK;

int gap = bits_hard_limit - bits;
// If there exists a risk of surpassing
// the parameter bits_hard_limit...
if (gap < MAXIMUM_WORST_CASE_EXCESS) {
  bool may_exceed_gap = true;
  if (allocated_bits >= MINIMUM_BITS_FOR_LINEAR_EXCESS_BOUNDING) {
    // In this case, the maximum possible difference between
    // the code lenth of an encoding of a vector s and bits,
    // is upper-bounded by a linear function a * bits + b
    int excess_upper_bound =
      (A_COEFFICIENT_FOR_LINEAR_EXCESS_BOUNDING * bits +
       B_COEFFICIENT_FOR_LINEAR_EXCESS_BOUNDING + FIXED_POINT_ROUNDING) >>
       FIXED_POINT_RESOLUTION;
    if (gap > excess_upper_bound) {
      may_exceed_gap = false;
    }
  }
  else if (N <= maximum_N_for_linear_excess_bounding) {
    // In this case, the maximum possible difference between
    // the code lenth of an encoding of a vector s and bits,
    // among all vectors in S(N, K), 2 <= N < 8,
    // is upper-bounded by a linear function (N - 1) / 2
    int excess_upper_bound = (N - 1) / 2;
    if (gap > excess_upper_bound) {
      may_exceed_gap = false;
    }
  }

  // If there exists still a risk of surpassing

  if (may_exceed_gap) {

      int bits_for_signs = (N < K) ? N : K;

      if (bits_for_signs >= remaining_bits) {
          // Recall that bits_hard_limit >= 2 by a condition checked above
          K = bits_hard_limit - 1;
```

Syntax	Reference

bits_for_signs = K;
}
int bits_for_index = remaining_bits - bits_for_signs;
 	
// if bits_for_index >= NUM_INDEX_BITS,
// then there is no risk of surpassing remaining_bits
// otherwise ...
if (bits_for_index < NUM_INDEX_BITS) {
uint bit_mask = 0xffffffff;
bit_mask >>= NUM_INDEX_BITS - bits_for_index;
 	
// bit_mask_complement has zeros in the
// bits_for_index less significant bits and ones in
// the remaining most significant bits.
uint bit_mask_complement = ~bit_mask;
 	
// Decrement K until the representation of an index in an enumeration of
// S(N,K) fits in bit_mask_complement.
// While K > N, the maximum number of bits for signs remains unchanged because
// min{N, K} is N. Once K reaches N (eventually from the begining),
// then each time K is decremented, the maximum number of bits for signs
// is also decremented, so an extra bit for the encoding of the index is
// onbtained and the bit_mask_complement is shifted one position
while (K > N && ((pyramid(N, K)-1) & bit_mask_complement)) {
K--;
}
while (K > 0 && ((pyramid(N, K)-1) & bit_mask_complement)) {
K--;
bit_mask_complement <<= 1;
}
}
}
}
 	
return K;
 	
}
	9.10.4.9.3
	
	9.10.4.9.3

9.10.4.9.5 max_k_for_n

The function *max_k_for_n(uint n)* computes the maximum value for *k* such that *pyramid(n,k)* fits in 32 bits.

Table 9-108: max_k_for_n

Syntax	Reference

// Compute value for K such pyramid(N,K) fits in 32 bits
//
// parameter[in] n: length of segment to be decoded
// return: the maximum value of k for given n.
 	
int max_k_for_n(uint n) {
if ((n < 1) || (n > MAX_BAND_SIZE) {
return -1;
}
return MAX_K_FOR_N[n-1];
}
	10.7.1
	
	10.7.3

9.10.4.9.6 hamming

The function *hamming()* computes the number of non-zero elements in a vector.

Table 9-109: hamming

Syntax

// parameter[in] N: vector length
// parameter[in] vector: input vector
// return: number of non-zero entries
 	
uint hamming(uint N, int vector[]) {

Syntax

uint length = 0;
 	
for (int i=0;i<N;i++) {
  if (vector[i] != 0) {
  length++;
  }
}
return length;
}

9.10.4.9.7 normalize

The function *normalize()* normalizes a vector vector to specified gain value *gain*.

Table 9-110: normalize

Syntax

// parameter[in]: length of vector
// parameter[in]: gain of the vector after normalization
// parameter[in/out]: vector to be normalized to specified gain
 	
void normalize(uint N, float gain, float vector[]) {
 	
float L2norm = 0;
for (int i=0;i<N;i++) {
L2norm += vector[i]*vector[i];
}
L2norm = sqrt(L2norm);
 	
float factor = gain / L2norm;
 	
for (int i=0;i<N;i++) {
vector[i] *= factor;
}
}

9.10.5 Frequency Transforms

9.10.5.1 Overview of Frequency Transforms

The spectral coefficients recovered from the bitstream are converted to time-domain samples using an overlapped inverse MDCT transform, implemented as a block DCT4 transform followed by phase-shifted window transform. The DCT4 transform operates directly on the band data as a single block transform, applied to the whole band (a single 1024-DCT4 transform [long]) or as a sequence of smaller transforms (8 128-DCT4 transforms [short]). The DCT4 transform is followed by a window transform, implementing a windowed overlap-add. This windowing operation will complete the transformation of the previous frame to the time-domain; the current frame will have the second part of the last transform block incomplete, until the next frame is available. This part of the processing chain is responsible for the 1-frame coding delay of the ACE decoder.

9.10.5.2 do_windowed_imdct

The windowed IMDCT function operates on a previous and current vector. After completion, the previous vector is ready for further processing, whereas the current vector will only be ready when the next frame has become available. This delay results in a one frame delay for the ACE codec output.

The function *do_windowed_imdct()* implements the inverse windowed IMDCT is shown in Table 9-111.

Table 9-111: do_windowed_imdct

Syntax	Reference

`// compute windowed imdct`
`//`
`// parameter[in] length: common length of previous and current vector`
`// parameter[in] N_prev: transform size for previous vector`
`// parameter[in] N_curr: transform size for current vector`
`// parameter[in/out]: previous vector with sample values`
`// parameter[in/out]: current vector with sample values`
 	
`void do_windowed_imdct(uint length,`
`uint N_prev,`
`uint N_curr,`
`float prev_vector[],`
`float curr_vector[]) {`
 	
`uint prev_window[];`
`uint curr_window[];`
 	
`uint K = min(N_prev,N_curr);`
`make_window(N_prev,K,prev_window);`
`make_window(N_curr,K,curr_window);`
 	
`do_multiple_idct4(N_cur, length, curr_vector);`
`do_multipe_window(N_curr,length,prev_window,curr_window,prev_vector,curr_vector);`
 	
`}`
	9.10.5.3.1
	9.10.5.3.1
	
	9.10.5.3.2
	9.10.5.3.4

9.10.5.3 Functions Supporting do_windowed_imdct

9.10.5.3.1 make_window

The function *make_window()* populates a window array *window* according to parameters *N* and *K*, where the former is the length of the half-window *window* and the latter is the size of the transition region from 0 to 1.

Table 9-112: make_window

Syntax

`// parameter[in] N: size of half-window`
`// parameter[in] K: size of transition region: (N-K)/2 boundary values at the left`
`//  are set to 0, and on the right are set to 1.`
`// parameter[out] window: array holding the half-window values`
 	
`// assumption: K <= N`
`// assumption: N, K are even`
 	
`void make_window(uint N, uint K, float window[]) {`
 	
`uint N_K_2 = (N-K)/2;`
`const float PI_2 = acos(0);`
 	
`for (int k=0;k<N_K_2;k++) {`
`window[k] = 0;`
`window[N-k-1] = 1;`
`}`
`for (k=0;k<K;k++) {`
`float W = sin(PI_2*(k+0.5)/K);`
`window[N_2] = sin(PI_2*W*W);`
`}`
`}`

9.10.5.3.2 do_multiple_idct4

The *do_multiple_idct4()* function applies the DCT4 transform to contiguous blocks within a vector *vector*.

Table 9-113: do_multiple_idct4

Syntax	Reference

`// parameter[in] N: the size of the DCT4 transform`
`// parameter[in] length: the length of the input vector`
`// parameter[in/out]: the vector to be transformed.`
`// assumption: N is a power of 2.`
 	
`void do_multiple_idct4(uint N, uint length, float vector[]) {`
`  uint num_blocks = length / N;`
`  for (int blk = 0; block<num_blocks;blk++) {`
`    do_single_idtc4(N,vector+blk*N);`
`  }`
`}`
	9.10.5.3.3

9.10.5.3.3 do_single_idct4

The *do_single_idct4()* function applies the DCT4 transform to an input vector *vector*. The DCT4 transform is a building block of the MDCT transform.

Table 9-114: do_single_idct4

Syntax

`// parameter[in] N: the size of the DCT4 transform`
`// parameter[in/out]: the vector to transformed.`
`// assumption: N is a power of 2.`
 	
`void do_single_idct4(uint N, float vector[]) {`
`  const float PI_4 = atan(1.0);`
`  float scratch_vector[];`
`  for (int m=0;m<N;m++) {`
`    ovector[m] = 0;`
`    for (int n=0;n<N;n++) {`
`      ovector[m] -= sqrt(2/N) * cos((PI_4/N)*(2*m+1)*(2*n+1)) * vector[size-1-n];`
`    }`
`  }`
`  for (int m=0;m<N;m++) {`
`    vector[m] = ovector[m];`
`  }`
`}`

9.10.5.3.4 do_multiple_window

The function *do_multiple_window()* applies a windowing with *prev_window* to the boundary between *prev_vector* and *curr_vector* and with *curr_window* to the non-boundary samples of *curr_vector*.

Table 9-115: do_multiple_window

Syntax	Reference

`// parameter[in] N: the number of samples (size) of the half-window`
`// parameter[in] length: the (common)length of prev_vector and curr_vector`
`// parameter[in] prev_window: the sample values of the half-window`
`//  applied to the boundary between prev_vector`
`//  and curr_vector`
`// parameter[in] curr_window: the sample values of the half-window`
`//  applied to the non-boundary samples of curr_vector`
`// parameter[in/out] prev_vector: the previous vector in the processing chain`
`// parameter[in/out] curr_vector: the current vector in the processing chain`
 	
`void do_multiple_window(uint N,`
`  uint length,`
`  float prev_window[],`
`  float curr_window[],`
`  float prev_vector,`
`  float curr_vector) {`
 	
`  float vector[];`
`  uint N_2 = N/2;`
 	
`  // copy from tail of prev_vector`	

Syntax	Reference

for (int n=0; n<N/2;n++) {
vector[n] = prev_vector[length-N_2+n];
}
// copy from curr_vector
for (int n=0;n<N;n++) {
vector[N_2+n] = curr_vector[n];
}
 	
uint num_blocks = length / N;
do_single_iwindow(N,length,prev_window,vector);
for (int blk=1;blk<num_blocks;blk++) {
do_single_window(N,length,curr_window,vector+blk*N);
}
 	
// copy to tail of prev_vector
for (int n=0; n<N/2;n++) {
prev_vector[length-N_2+n] = vector[n];
}
// copy to curr_vector
for (int n=0;n<N;n++) {
curr_vector[n] = vector[N_2+n];
}
}
	9.10.5.3.5

9.10.5.3.5 do_single_window

The *do_single_window()* function applies the half-window *window* to the vector *vector*. After application, the left half of vector has been transformed corresponding to the mirrored version of *window* (i.e. the falling part of a complete window) and the right half of *vector* has been transformed to *window* as the left half of the a complete window.

Table 9-116: do_single_window

Syntax

// parameter[in]N: the number of samples (size) of the half-window
// parameter[in]window: the sample values of the rising left half-window
// parameter[in/out] vector: the vector to which windowing is applied
 	
void do_single_window(uint N, float window[], float vector[]) {
 	
for (int n=0;n<N/2;n++) {
float v0;
float v1;
v0 = +window[N-n-1]*vector[n] - window[n]*vector[N-n-1];
v1 = +window[n]*vector[n] + window[N-n-1]*vector[N-n-1];
vector[n] = v0;
vector[N-n-1] = v1;
}
}

9.10.5.4 Band Reshaping

The function *reshape_band()* reshapes the time-frequency representation of a vector *vector* from an original number of time slots *old_num_blocks* to a new number of time slots *new_num_blocks*. The in/out vector holds the values for a given frequency band, with start bin *band_start* (included) and end bin *band_end* (excluded). The start and end values are relative to a frequency transform with size *frame_duration*.

Table 9-117: reshape_band

Syntax	Reference

void reshape_band(uint frame_duration,
uint band_start,
uint band_end,
uint old_num_blocks,
uint new_num_blocks,
float vector[]) {
 	
uint num_blocks = old_num_blocks;	

Syntax	Reference

uint actual_band_start = band_start / num_blocks;
uint actual_band_end = band_end / num_blocks;
uint num_samples_band_block = actual_band_end - actual_band_start;
uint num_samples_block = frame_duration / num_blocks;
 	
// "fake" frame (zero padding to prevent a transient on startup)
float frame[];
set (frame,0);
 	
// insert bands in "fake" frame and convert frame to time domain
for (uint blk=0; blk < num_blocks;blk++) {
 	
float band_block = vector + blk*num_samples_band_block;
float block = frame + blk* num_samples_block;
 	
// insert band in band_block
for (uint n=0;n<<num_samples_band_block;n++) {
block[actual_band_start+n] = band_block[n];
}
}
 	
// convert frame fully to time domain
do_multiple_haar(num_samples_block,frame_duration,frame);
	9.10.5.5

num_blocks = new_num_blocks;
 	
uint actual_band_start = band_start / num_blocks;
uint actual_band_end = band_end / num_blocks;
uint num_samples_band_block = actual_band_end - actual_band_start;
uint num_samples_block = frame_duration / num_blocks;
 	
// convert frame to frequency domain
do_multiple_haar(num_samples_block,frame_duration,frame);
	9.10.5.5

// extract bands from "fake" frame
for (uint blk=0; blk < num_blocks;blk++) {
 	
float band_block = vector + blk*num_samples_band_block;
float block = frame + blk* num_samples_block;
 	
// insert band in band_block
for (uint n=0;n<<num_samples_band_block;n++) {
band_block[n] = block[actual_band_start+n];
}
}
}	

9.10.5.5 Haar Transform

9.10.5.5.1 Overview of Haar Transform

The Haar transform used is the sequentially ordered Haar transform used for reconstruction from time-frequency reshaping in the ACE encoder.

9.10.5.5.2 do_multiple_haar

The *do_multiple_haar()* function applies the Haar transform to contiguous blocks within a vector *vector*.

Table 9-118: do_multiple_haar

Syntax	Reference

`// parameter[in] N: the size of the Harr transform`
`// parameter[in] length: the length of the input vector`
`// parameter[in/out]: the vector to transformed.`
`// assumption: N is a power of 2.`
 	
`void do_multiple_haar(uint N, uint length, float vector[]) {`
`  uint num_blocks = length / N;`
`  for (int blk = 0; block<num_blocks;blk++) {`
`  do_single_haar(N,vector+blk*N);`
`  }`
`}`
	9.10.5.5.3

9.10.5.5.3 do_single_haar

The *do_single_haar()* function applies the Haar transform to an input vector *vector*.

Table 9-119: do_single_haar

Syntax	Reference

`// parameter[in] N: the size of the Haar transform`
`// parameter[in/out]: the vector to be transformed.`
`// assumption: N is a power of 2.`
 	
`void do_single_haar(N, float vector[]) {`
 	
`  float scratch_vector[];`
 	
`  for (int m=0;m<N;m++) {`
`  ovector[m] = 0;`
`  for (int n=0;n<N;n++) {`
`  ovector[m] = haar_coefficient(N,m,n) *vector[n];`
`  }`
`  }`
 	
`  for (int m=0;m<N;m++) {`
`  vector[m] = ovector[m] / sqrt(N);`
`  }`
`}`
`  for (int m=0;m<N;m++) {`
`  vector[m] = ovector[m];`
`  }`
`}`
	9.10.5.5.4

9.10.5.5.4 haar_coefficient

The function *haar_coefficient()* computes the index for pair *(m,n)* the corresponding entry (either -1 or +1) in the Haar transform. The final result is normalized by a factor *sqrt(N)*.

Table 9-120: haar_coefficient

Syntax

```
// parameter[in] N: the size of the sequentially ordered transform
// parameter[in] m: row index
// parameter[in] n: column index
// return: the entry for index pair(m,n)

// assumption: N is a power of 2.

float haar_coefficient(uint  N, uint m, uint n) {

  uint M = log2(N);

  m = m & (N - 1);
  n = n & (N - 1);

  float sum =0;
  for (int k=0;k<M;k++) {
    sum += ((m>>k)*(n>>(M-1-k)))&1;
  }
  for (int k=1;k<M;k++) {
    sum += ((m>>k)*(n>>(M-k)))&1;
  }
  return (sum&1) ? -1 : 1;
}
```

9.10.6 Long Term Synthesis (LTS)

9.10.6.1 Overview of LTS

The long term synthesis module restores 'predictable' signal components that have been removed as a preprocessing step at encoding time. The missing signal components are restored using 1 024 samples previously decoded. A current buffer of samples is restored by time-varying IIR filtering.

9.10.6.2 LTS Constants and Definitions

The list of constants and definitions shared by all LTS modules is shown in Table 9-121.

Table 9-121: LTS constants

Syntax

```
// Pi
const float PI_2 = acos(0);

// Length of history buffer
const uint LTS_PREDICTION_RANGE = 1024;

// Length of transition window
const uint WINDOW_TRANSITION = 102;

// Thresholds for window decision
const float WINDOW_THRSH_LOW = 0.25;
const float WINDOW_THRSH_MID = 0.50;

// Window shape parameters
const float WINDOW_ALPHA_LOW = 1.75;
const float WINDOW_ALPHA_MID = 2.00;
const float WINDOW_ALPHA_HGH = 2.50;
```

9.10.6.3 do_longterm_synthesis

The function *do_longterm_synthesis()*, shown in Table 9-122, is the top level function for long term synthesis.

Table 9-122: do_longterm_synthesis

Syntax	Reference

`// Apply Long Term Synthesis to buffer of time samples in current frame.`
`//`
`// parameter[in] pcm1: the corresponding buffer of time samples in the`
`//  previous frame`
`// parameter[in/out] pcm0: a buffer of time samples in the current frame.`
 	
`do_longterm_synthesis(float pcm2[], float pcm1[], float pcm0[]) {`
 	
`  // prediction filter lag for previous and current frame`
`  uint curr_lag = _lts_lag;`
`  uint prev_lag = _previous->_lts_lag;`
 	
`  // filter coefficients for previous and current frame`
`  float curr_gains[] = _lts_coefficients;`
`  float prev_gains[] = _previous->_coefficients;`
 	
`  // concatenate buffers and point to the position of the current frame`
`  float lts_buffer[];`
`  lts_buffer += LTS_PREDICTION_RANGE;`
`  lts_make_buffer(pcm2, pcm1, pcm0, lts_buffer);`
	9.10.6.4.1

`  // set windows for current and previous frames`
`  float curr_window[], prev_window[];`
`  set(curr_window,1,frame_duration);`
`  set(prev_window,0,frame_duration);`
`  lts_set_windows(prev_lag, curr_lag,`
`  prev_gains,curr_gains,`
`  prev_window,curr_window);`
	9.10.6.4.2

`  // update copy of current frame from previous samples`
`  lts_restore(prev_lag, curr_lag,`
`  prev_gains, curr_gains,`
`  prev_window, curr_window,`
`  lts_buffer);`
 	
`  // copy result to current buffer`
`  for (int i=0;i< frame_duration;int) {`
`  pcm0[i] = lts_buffer[i];`
`  }`
`}`
	9.10.6.4.3

9.10.6.4 Functions Supporting do_longterm_synthesis

9.10.6.4.1 lts_make_buffer

The function *lts_make_buffer()* concatenates the current buffer of samples with the corresponding buffer of one or two previous frames.

Table 9-123: lts_make_buffer

Syntax

`// Concatenate three input buffers in a single buffer.`
`//`
`// parameter[in] pcm2: sample array corresponding to twice previous frame`
`// parameter[in] pcm1: sample array corresponding to previous frame`
`// parameter[in] pcm0: sample array for current frame`
`// parameter[out] lts_buffer: concatenated buffer`
 	
`void lts_make_buffer(float pcm2[], float pcm1[], float pcm0[], float lts_buffer)  {`
 	
`  for (int i=0;i< frame_duration;int) {`
`  lts_buffer[i] = pcm0[i];`
`  }`
 	
`  for (int i=0;(i<frame_duration) &&`
`  (frame_duration <= (i + LTS_PREDICTION_RANGE));i++) {`
`  lts_buffer[i-frame_durararion] = pcm1[i];`
`  }`
 	
`  for (int i=0;(i<frame_duration) &&`

Syntax

(2*frame_duration <= (i + LTS_PREDICTION_RANGE));i++) {
lts_buffer[i-2*frame_duraration] = pcm2[i];
}
}

9.10.6.4.2 lts_set_windows

The function *lts_set_windows()* creates the fade-in / fade-out windows for transitioning from the set of prediction parameters corresponding to the previous frame to the set of parameters for the current frame. The windows are constant 0 (previous frame) or 1 (current frame) except in the transition region of size *WINDOW_TRANSITION*.

Table 9-124: lts_set_windows

Syntax

// Create fade-in / fade-out windows
//
// parameter[in] prev_lag: filter lag for previous frame.
// parameter[in] curr_lag: filter lag for current frame.
// parameter[in] prev_gains: filter coefficients for previous frame.
// parameter[in] curr_gains: filter coefficients for current frame.
// parameter[in/out] prev_window: fade-out window for previous frame.
// parameter[in/out] curr_window: fade-in window for current frame.
 	
void lts_set_windows(prev_lag, curr_lag,
prev_gains,curr_gains,
prev_window, cur_window) {
 	
float diff_gain = abs(curr_gains[0] - prev_gains[0]);
float alpha = WINDOW_ALPHA_MID;
 	
if (prev_lag == curr_lag) {
if (gain_diff < WINDOW_THRSH_LOW) {
alpha = WINDOW_ALPHA_LOW;
}
else if (gain_diff < WINDOW_THRSH_MID) {
alpha = WINDOW_ALPHA_MID;
}
else {
alpha = WINDOW_ALPHA_HGH;
}
}
 	
for (int i=0;i<WINDOW_TRANSITION;i++) {
float window_value = sin(PI_2 * (i+0.5) / WINDOW_TRANSITION);
window_value = pow(window_value,alpha);
prev_window[i] = 1 - window_value;
curr_window[i] = window_value;
}
}

9.10.6.4.3 lts_restore

The function *lts_restore()* adds the missing components to an array of samples that have been removed at encoding time.

Table 9-125: lts_restore

Syntax

```
// restore missing components by long term synthesis
//
// parameter[in] prev_lag: filter lag for previous frame.
// parameter[in] curr_lag: filter lag for current frame.
// parameter[in] prev_gains: filter coefficients for previous frame.
// parameter[in] curr_gains: filter coefficients for current frame.
// parameter[in] prev_window: fade-out window for previous frame.
// parameter[in] curr_window: fade-in window for current frame.
// parameter[in/out] lts_buffer: sample array to be restored.

void lts_restore(float prev_lag, float curr_lag,
                 float prev_gains[],float curr_gains[],
                 float prev_window[],float curr_window[],
                 float lts_buffer[]) {

  for (int i=0;i<frame_duration;i++) {

    lts_buffer[i] += prev_window[i] * (lts_buffer[prev_lag-2] + lts_buffer[prev_lag+2]) *
                                                                    prev_gains[2];
    lts_buffer[i] += prev_window[i] * (lts_buffer[prev_lag-1] + lts_buffer[prev_lag+1]) *
                                                                     prev_gains[1];
    lts_buffer[i] += prev_window[i] * lts_buffer[prev_lag] * prev_gains[0];
    lts_buffer[i] += curr_window[i] * (lts_buffer[curr_lag-2] + lts_buffer[curr_lag+2]) *
                                                                      curr_gains[2];
    lts_buffer[i] += curr_window[i] * (lts_buffer[curr_lag-1] + lts_buffer[curr_lag+1]) *
                                                                      curr_gains[1];
    lts_buffer[i] += curr_window[i] * lts_buffer[curr_lag] * curr_gains[0];

  }
}
```

9.10.7 Temporal Hole Fill (informative)

9.10.7.1 Overview of Temporal Hole Filling

Temporal Hole Filling (THF) is an optional part of ACE Stream decoding and applies only in case of a short transform (i.e. 8 MDCT blocks within a frame). Temporal Hole Filling is any process that injects energy in all-zero bands of an MDCT block, in combination with rescaling the modified band to the original norm. The encoder indicates to the decoder using the *_thf_enabled* flag whether or not temporal hole filling is appropriate. The *_thf_enabled* flag, when set to *true*, indicates that an insufficient number of bits were available to encode all MDCT blocks.

The energy injection method is not normative. A decoder is free to choose any method for energy injection, as long as the modified band is rescaled to its original norm. The pseudo-code below provides an example of temporal hole filling.

9.10.7.2 do_temporal_hole_fill (informative)

The function *do_temporal_hole_fill()*, in Table 9-126, runs over all effective channels and all effective bands and applies temporal hole filling. This function only has effect when temporal hole filling is enabled and only applies when *_is_short* is true.

Table 9-126: do_temporal_hole_fill

Syntax	Reference

```
// Temporal Hole Filling
// Apply temporal hole filling across all channels and bands

float do_temporal_hole_fill() {
  for (channel=_first_effective_channel;channel<_last_effective_channel;channel++) {
    for (uint band=0;band<_num_effective_bands;band++) {
      do_band_temporal_hole_fill(channel,band);
    }
  }
}
```

9.10.7.3

9.10.7.3 do_band_temporal_hole_fill (informative)

The function *do_band_temporal_hole_fill()* in Table 9-127 injects energy into an all-zero band of a short MDCT block.

Table 9-127: do_band_temporal_hole_fill

Syntax	Reference

`// Temporal hole filling (THF)`
`// Signal energy is inserted in all-zero blocks.`
`// This processing only applies to short frames.`
`//`
`// parameter[in/out] buffer: buffer with`
 	
`void do_band_temporal_hole_fill(uint channel, uint band) {`
 	
`  const uint num_blocks = 8;`
`  const uint block_size = 128;`
 	
`  float norm = pow(2.0,_lognorm[channel][band]);`
`  uint num_samples = _config->bandsize[band];`
`  uint num_channels = _num_effective_channels;`
`  float band_buffer[] = _stream_buffer[channel] + _config->band_boundaries[band];`
 	
`  float bits_per_bin = (float) _alloc[band] / (num_samples *  num_channels);`
`  float factor = pow(2,-2.0 * bits_per_bin);`
 	
`  if (_is_short & _thf_enabled) {`
 	
`    float block_buffer[] = band_buffer;`
`    for (int block = 0; block < 8; block++) {`
`      if (is_zero_block(block_buffer,128)) {`
`        for (int n=0;n < block_size;n++) {`
`          block_buffer[n] = random_sign() * norm * factor / sqrt(num_samples);`
`        }`
`      }`
`      block_buffer += 128;`
`    }`
`    normalize(num_samples,norm,band_buffer);`
`  }`
`}`
	9.10.7.4
	
	9.10.4.9.7

9.10.7.4 random_sign

The function *random_sign()*, shown in Table 9-128, draws with equal probability between the values -1 and +1.

Table 9-128: random_sign

Syntax	Reference

`// random equal probability drawing of -1 and +1`
 	
`int random_sign() {`
 	
`  bool result = -1;`
 	
`  float value = Rand(0.0,2.0);`
`  if (value >= 1.0) {`
`    result = +1;`
`  }`
`  return result;`
`}`
	8.3.6

9.10.8 Deemphasis

9.10.8.1 Overview of Deemphasis

A final step in the decoding process boosts lower frequencies and attenuates higher frequencies, compensating for the inverse pre-processing step in the encoding process. The deemphasis filter is implemented as a one-tap IIR filter. The filter state is maintained in the *_deemphasis_prev* variable.

9.10.8.2 do_deemphasis

The function *de_demphasis()* shown in Table 9-129, applies a demphasis filter to all output streams of an ACE stream decoder.

Table 9-129: do_deemphasis

Syntax	Reference

`// Deemphasis filtering over all channels`
`// updating state for the next frame.`
 	
`void do_demphasis() {`
 	
`  for (int channel = 0 ; channel < _num_stream_channels; channel++) {`
`    float prev = 0;`
`    if (_previous) {`
`      prev = _previous->_deemphasis_prev[channel];`
`    }`
`    _deemphasis_prev[channel] = do_channel_deemphasis(prev,_stream_buffer[channel]);`
`  }`
`}`
	9.10.8.3.1

9.10.8.3 Function Called by do_deemphasis

9.10.8.3.1 do_channel_deemphasis

The function *do_channel_deemphasis(float prev, float buffer[])* operates on a single channel, boosting lower frequencies and attenuating higher frequencies.

Table 9-130: do_channel_deemphasis

Syntax

`// Deemphasis filtering for one channel`
`//`
`// Low frequency boost / high frequency attenuation filtering`
`// (+16.5dB @ 100Hz, 0dB @ 8.65kHz, -5dB @ 20kHz)`
`//`
`// parameter[in] prev: filter state`
`// parameter[in/out] buffer: sample buffer`
`// return: filter state`
 	
`float do_channel_deemphasis(float prev, float buffer[]) {`
 	
`  const float DEEMPHASIS_COEFFICIENT = 0.85;`
 	
`  for (int n=0; n < _frame_size;n++) {`
`    buffer[n] += prev;`
`    prev = DEEMPHASIS_COEFFICIENT * buffer[n];`
`  }`
`  return prev;`
`}`

9.11 LFE Decoding

9.11.1 Overview of LFE Decoding

An LFE channel is reconstructed in two steps:

1) reconstruction of a decimated representation; and

2) upsampling to full sampling frequency.

Reconstruction optionally involves insertion of synthetic noise. An informative example for random noise generation is included for completeness.

All functions in this clause have access to all the member variables of the relevant LFE channel instance.

9.11.2 LFE Constants and Definition

The constants defined in Table 9-131 are used throughout the remainder of the LFE pseudo-code examples.

Table 9-131: LFE Constants

LFE Constants and DefinitionSyntax

`const uint kLFEAbsoluteMode = 0;  // LFE samples at full resolution`
`const uint kLFEPredictMode = 1;  // LFE samples as residuals`
`const uint kLFEReducedMode = 2;  // LFE sampels as reduced resolution`
`// residuals`
`const uint kLFESyntheticMode = 3;  // LFE samples as synthetic noise`
 	
`const uint LFE_INTERPOLATION_FACTOR = 64;  // LFE interpolation factor`
`const uint LFE_INTERP_FILTER_LENGTH = 1024; // Length of LFE interpolation filter`

9.11.3 decode_lfe_channel

decode_lfe_channel(), shown in Table 9-132 reconstructs an LFE channel in two steps. In the first step a subsampled version of the LFE channel is reconstructed from the data read from the bitstream, either predictively (*kLFEPredictMode*) or absolutely (*kLFEAbsoluteMode*, *kLFEReducedMode*). If the *lfe_mode* is given as *kLFESyntheticMode*, the subsampled value may optionally synthetically constructed. The method of generating synthetic noise is not normative, with an informative example given in clause 9.11.4.3. In a second step, the subsampled channel is upsampled to the full sampling frequency.

Table 9-132: decode_lfe_channel

Syntax	Reference
`// Reconstruct LFE channel`	
`decode_lfe_channel() {`	

`  if (_lfe_mode == kPredictMode) {`
`  lfe_predict();`
`  }`
	9.11.4.1

`  else if ((_lfe_mode == kAbsoluteMode) || (lfe_mode == kReducedMode)) }`
`  lfe_reconstruct();`
`  }`
	9.11.4.2

`  else {`
`  lfe_synthesize();`
`  }`
	9.11.4.3

`  // extend previous LFE decimated channel on the right`
`  float prev_lfe_decimated_channel[] = previous-> _lfe_decimated_channel;`
`  for (int i=0;i<_decimated_frame_duration;i++) {`
`  prev_lfe_decimated_channel[_decimated_frame_duration+i] = _lfe_decimated_channel[i];`
`  }`	

`  // upsample previous decimated channel`
`  previous->lfe_upsample();`
`}`
	9.11.4.4

9.11.4 Functions Supporting decode_lfe_channel

9.11.4.1 lfe_predict

The function *lfe_predict()* reconstructs the LFE decimated channel from the residual sample values and prediction coefficients that have been read from the bitstream.

Table 9-133: lfe_predict

Syntax

`// Reconstruct the LFE decimated channel in predictive mode`
 	
`lfe_predict() {`
 	
`  _lfe_decimated_channel[-1] = previous->_lfe_decimated_channel[_decimated_frame_duration-1];`
`  _lfe_decimated_channel[-2] = previous->_lfe_decimated_channel[_decimated_frame_duration-2];`
 	
`  for (int i=0;i<_decimated_frame_duration;i++) {`
 	
`    _lfe_decimated_channel[i] = _lfe_decimated_channel[i]*_stepsize +`
`                  _lfe_predictor[0]* _lfe_decimated_channel[i-1] +`
`                  _lfe_predictor[1]* _lfe_decimated_channel[i-2];`
`  }`
`}`

9.11.4.2 lfe_reconstruct

The function *lfe_reconstruct()* reconstructs the LFE decimated channel from the quantized sample values that have been read from the bitstream.

Table 9-134: lfe_reconstruct

Syntax

`// Reconstruct LFE decimated channel from quantized sample values`
 	
`lfe_reconstruct() {`
`  for (int i=0;i<_decimated_frame_duration;i++) {`
 	
`    _lfe_decimated_channel [i] = _lfe_decimated_channel [i]*_stepsize;`
`  }`
`}`

9.11.4.3 lfe_synthesize (informative)

The function *lfe_synthesize()* inserts synthetic noise of the appropriate power as determined by the value of *db_norm*. Various methods of pseudo-random noise generation may be used here.

Table 9-135: lfe_synthesize

Syntax	Reference

`// Inject synthetic noise`
 	
`lfe_synthesize() {`
 	
`  const float LFE_NORM_FACTOR = 1.73;`
 	
`  for (int i=0;i<_decimated_frame_duration;i++) {`
 	
`    _lfe_decimated_channel[i] = Rand(-1.0,1.0) * LFE_NORM_FACTOR`
`/db_to_linear(db_norm);`
`  }`
`}`
	9.11.4.5

9.11.4.4 lfe_upsample

The function *lfe_upsample()* interpolates the LFE decimated channel to full frame duration. This function is called from the next (future) frame after extending the LFE decimated channel with current decimated sample values.

Table 9-136: lfe_upsample

Syntax	Reference

`// Interpolation of LFE decimated channel to full frame duration`
 	
`lfe_upsample() {`
 	
`  uint filter_history = LFE_INTERP_FILTER_LENGTH / LFE_INTERPOLATION_FACTOR;`
 	
`  // Loop over interpolation blocks`
`  for (int n=0;n<_decimated_frame_duration;n++) {`
 	
`    // Loop over samples in interpolation block`
`    for (int i=0; i<LFE_INTERPOLATION_FACTOR;k++ ) {`
 	
`      // Loop over filter coefficients`
`      _lfe_channel[n*LFE_INTERPOLATION_FACTOR+k] = 0;`
`      for (k=0; k < filter_history ; k++) {`
`        float coeff = INTERPOLATION_FILTER_TABLE[i+k*LFE_INTERPOLATION_FACTOR];`
`        float sample = _lfe_decimated_channel[n + filter_history - 1 - k];`
`        _lfe_channel[n*LFE_INTERPOLATION_FACTOR+i] += coeff*sample;`
`      }`
`    }`
`  }`
`}`
	10.8.1

9.11.4.5 db_to_linear

The function *db_to_linear(int n)* converts a integer dB value to a linear (floating point) scalar.

Table 9-137: db_to_linear

Syntax

`// Converting integer dB to linear`
`//`
`// parameter[in] n: input value`
`// return 10^(val/10)`
 	
`float db_to_linear(int n) {`
 	
`    float crt2[] =  {1.0, 1.25992107, 1.58740103};`
 	
`    int prem = (n % 3 + 3) % 3;`
`    int exp = (n - prem) / 3;`
 	
`    return ldexp(crt2[prem], exp);`
`}`

10 Tables and Constants

10.1 Overview

The clauses throughout clause 10 contain constants and look-up tables used by the ACE bitstream extraction and decoding algorithms in clause 9.

10.2 System Constants and Tables

10.2.1 Basic Constants

Table 10-1: System Constants

Syntax
`const uint NUM_ACE_BANDS = 22;  // Number of spectral bands`

10.2.2 Coding Modes

Table 10-2: Coding Modes

Syntax

`const uint kCodingFull = 0;`
`const uint kCodingOff = 1`
`const uint kCodingLeft = 2;`
`const uint kCodingRight = 3;`

10.2.3 BandsConfig

10.2.3.1 Structure BandsConfig

The *BandsConfig* datatype defines the basic aspects of spectral representation.

Table 10-3: BandsConfig

Syntax

`struct BandsConfig {`
 	
`  const uint num_bands;  // number of bands`
`  const uint index_largest_band;  // index of the largest band`
`  const uint num_narrow_bands;  // number of minimal size bands`
`  const band_boundaries[NUM_ACE_BANDS+1];  // band boundaries`
`  const LBFP log_band_size[NUM_ACE_BANDS];  // log2 of the band size`
`  const LNFP lognorm_offset[NUM_ACE_BANDS]  // an offset on lognorms used in lognorm prediction`
`  const uint alloc_max[NUM_ACE_BANDS];  // max bit allocations for bands`
`  const uint alloc_min[NUM_ACE_BANDS];  // min bit allocations for bands`
`  const LNFP refinement_delta[NUM_ACE_BANDS]; // adjustment per band used in refinement allocation`
`}`

Parameters

- **num_bands** - Number of spectral bands
- **index_largest_band** - Band with the largest number of spectral lines
- **band_boundaries** - Start index for band
- **band_size** - Band size
- **log_band_size** - Log band size
- **lognorm_offset** - Bitstream encoding lognorm offset
- **alloc_max** - Maximum bit allocation per band
- **refinement_delta** - Initial lognorm refinement values

10.2.3.2 Bands Configuration Tables

10.2.3.2.1 bands_config_48khz

Table 10-4 lists the configuration tables for 48 KHz.

Table 10-4: bands_config_48khz

Syntax

```
const BandsConfig BANDS_CONFIG_48KHZ = {
  // num_bands
  22,
  // index_largest_band
  21,
  // num_narrow_bands
   8,
  // band_boundaries
  {
    0, 8, 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 144,
    176, 208, 240, 288, 336, 416, 512, 672, 848, 1024
  },
  // bandsize
  {
    8, 8, 8, 8, 8, 8, 8, 8, 16, 16, 16, 32, 32, 32, 32, 48, 48, 80, 96, 160, 176, 176
  }
  // log_band_size
  {
    3.00000000, 3.00000000, 3.00000000, 3.00000000, 3.00000000, 3.00000000, 3.00000000,
    3.00000000, 4.00000000, 4.00000000, 4.00000000, 5.00000000, 5.00000000, 5.00000000,
    5.00000000, 5.56250000, 5.56250000, 6.31250000, 6.56250000, 7.31250000, 7.43750000,
    7.43750000
  },
  // lognorm offset
  {
     8.8554687500, 8.7666015625, 8.3984375000, 8.0214843750,
     7.8046875000, 7.5625000000, 7.4238281250, 7.3964843750,
     7.8808593750, 7.7509765625, 7.6406250000, 8.1269531250,
     7.9189453125, 7.7216796875, 7.5195312500, 7.7041015625,
     7.5351562500, 7.7236328125, 7.5419921875, 7.3388671875,
     6.4667968750, 5.6230468750
  },
  // alloc_max
  {
    68,  68,  68,  68,  68,  68, 68, 68, 136, 136, 136, 272, 272, 272, 272, 384,
   384, 592, 608, 608, 640, 640
   },
  // alloc_min
  {
    2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 6, 6, 6, 6, 8, 8, 14, 17, 28, 30, 30
  },
  // refinement_delta
  {
    -1.12500000, -1.12500000, -1.12500000, -1.12500000, -1.12500000, -1.12500000, -1.12500000,
    -1.12500000, -0.62500000, -0.62500000, -0.62500000, -0.12500000, -0.12500000, -0.12500000,
    -0.12500000, +0.12500000, +0.12500000, +0.50000000, +0.62500000, +1.00000000, +1.06250000,
    +1.06250000
  }
}
```

10.2.3.2.2 bands_config_44p1khz

Table 10-5 lists the configuration tables for 44,1 KHz.

Table 10-5: bands_config_44p1khz

Syntax

```
const BandsConfig BANDS_CONFIG_44P1KHZ = {

  // num_bands
  22,
  // index_largest_band
  20,
```

Syntax

```
  // num_narrow_bands
   6,
  // band_boundaries
  {
    0, 8, 16, 24, 32, 40, 48, 64, 80, 96, 112, 128,
    160, 192, 224, 272, 320, 368, 448, 560, 720, 928, 1024
  },
   // bandsize
  {
    8, 8, 8, 8, 8, 8, 16, 16, 16, 16, 16, 32, 32, 32, 48, 48, 48, 80, 112, 160, 208, 96
  }
  // log_band_size
  {
    3.00000000, 3.00000000, 3.00000000, 3.00000000, 3.00000000, 3.00000000, 3.00000000,
    4.00000000, 4.00000000, 5.00000000, 5.00000000, 5.00000000, 5.00000000, 5.00000000,
    5.56250000, 5.56250000, 5.56250000, 6.31250000, 6.81250000, 7.31250000, 7.68750000,
    6.56250000
  },
  // lognorm offset
  {
     8.8554687500, 8.7666015625, 8.3984375000, 8.0214843750,
     7.8046875000, 7.5625000000, 7.4238281250, 7.3964843750,
     7.8808593750, 7.7509765625, 7.6406250000, 8.1269531250,
     7.9189453125, 7.7216796875, 7.5195312500, 7.7041015625,
     7.5351562500, 7.7236328125, 7.5419921875, 7.3388671875,
     6.4667968750, 5.6230468750,
  },
  // alloc_max
  {
     68,  68,  68,  68,  68,  68, 136, 136, 136, 136, 136, 272, 272, 272, 408, 384,
    384, 592, 708, 608, 756, 348
  },

  // alloc_min
  {
    2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 6, 6, 6, 8, 8, 8, 14, 19, 28, 36, 17
  },

  // refinement_delta
  {
    -1.12500000, -1.12500000, -1.12500000, -1.12500000, -1.12500000, -1.12500000, -0.62500000,
    -0.62500000, -0.62500000, -0.62500000, -0.62500000, -0.12500000, -0.12500000, -0.12500000,
    +0.12500000, +0.12500000, +0.12500000, +0.50000000, +0.75000000, +1.00000000, +1.18750000,
    +0.62500000
  }
}
```

10.3 Long Term Synthesis Constants and Tables

10.3.1 LTS_FILTER_CODEBOOK

Table 10-6: LTS Filter Codebook

Syntax

```
// LTS_FILTER_CODEBOOK

float LTS_FILTER_CODEBOOK[29[3] =
{
  {0,           0,          0},
  {0.064995,    0.040695,   0.020557},
  {0.079613,    0.047869,   0.006069},
  {0.115744,    0.029569,   0.004913},
  {0.15375,     0.01125,    0.005625},
  {0.097493,    0.061043,   0.030836},
  {0.119419,    0.071803,   0.009104},
  {0.173616,    0.044353,   0.007369},
  {0.230625,    0.016875,   0.008437},
  {0.12999,     0.08139,    0.041115},
  {0.159225,    0.095737,   0.012139},
  {0.231487,    0.059138,   0.009825},
  {0.3075,      0.0225,     0.01125},
```

```
Syntax
  {0.162487,    0.101737,   0.051393},
  {0.199031,    0.119672,   0.015173},
  {0.289359,    0.073922,   0.012281},
  {0.384375,    0.028125,   0.014062},
  {0.194985,    0.122085,   0.061672},
  {0.238838,    0.143606,   0.018208},
  {0.347231,    0.088706,   0.014737},
  {0.46125,     0.03375,    0.016875},
  {0.227482,    0.142432,   0.071951},
  {0.278644,    0.167541,   0.021243},
  {0.405103,    0.103491,   0.017194},
  {0.538125,    0.039375,   0.019687},
  {0.25998,     0.16278,    0.082229},
  {0.31845,     0.191475,   0.024278},
  {0.462975,    0.118275,   0.01965},
  {0.615,       0.045,      0.0225}
}
```

10.4 Lognorm Tables and Constants

10.4.1 Lognorm Basic Constants

Table 10-7: Lognorm Constants

```
Syntax
const LNFP LOGNORM_MINUS_INFINITY = -32;
```

10.4.2 LOGNORM_GOLOMB_PARAMS

Table 10-8: Lognorm Golomb Parameters

```
Syntax
static const GolombCodeParams golomb_params_64_0_15 = { 64,  0, 15};
static const GolombCodeParams golomb_params_64_1_31 = { 64,  1, 31};
static const GolombCodeParams golomb_params_64_2_31 = { 64,  2, 31};
static const GolombCodeParams golomb_params_64_3_31 = { 64,  3, 31};

uint NUM_PRED_SETS = 2;
uint NUM_TRANSFORMS = 2;

const GolombCodeParams LOGNORM_GOLOMBS_PARAMS[NUM_PRED_SETS][NUM_TRANSFORMS][NUM_ACE_BANDS] =
{
  // predictive frame
  { // predictive frame, long transform
    {
      &golomb_params_64_1_31, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15,
    },
    // predictive frame, short transform
    {
      &golomb_params_64_1_31, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15,
    },
  },
  // non_predictive frame
  {
```

Syntax

```
    // non_predictive frame, long transform
    {
      &golomb_params_64_2_31, &golomb_params_64_1_31, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15,
    },
    // non_predictive frame, short transform
    {
      &golomb_params_64_3_31, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15, &golomb_params_64_0_15, &golomb_params_64_0_15,
      &golomb_params_64_0_15,
    },
  }
};
```

10.4.3 F_PREDICTOR_TABLE

Table 10-9: F Predictor Table

Syntax

```
uint const NUM_PRED_SETS = 2;

const LNFP F_PREDICTOR_TABLE[NUM_PRED_SETS][NUM_ACE_BANDS] =
{
  // predictive frame
  {
    0.5996093750, 0.5800781250, 0.5654296875, 0.5498046875,
    0.5371093750, 0.5253906250, 0.5146484375, 0.5048828125,
    0.5000000000, 0.5000000000, 0.5000000000, 0.5000000000,
    0.5000000000, 0.5000000000, 0.5000000000, 0.5000000000,
    0.5000000000, 0.5000000000, 0.5000000000, 0.5000000000,
    0.5000000000, 0.5000000000
  },
  // non_predictive frame
  {
     0.0000000000, 0.0000000000, 0.0000000000, 0.0000000000,
     0.0000000000, 0.0000000000, 0.0000000000, 0.0000000000,
     0.0000000000, 0.0000000000, 0.0000000000, 0.0000000000,
     0.0000000000, 0.0000000000, 0.0000000000, 0.0000000000,
     0.0000000000, 0.0000000000, 0.0000000000, 0.0000000000,
     0.0000000000, 0.0000000000
  }
};
```

10.4.4 B_PREDICTOR_TABLE

Table 10-10: B Predictor Table

Syntax

```
uint const NUM_PRED_SETS = 2;

const LNFP B_PREDICTOR_TABLE[NUM_PRED_SETS][NUM_ACE_BANDS] =
{
  // predictive frame
  {
    0.0000000000, 1.0000000000, 0.7500000000, 0.5000000000,
    0.5000000000, 0.4501953125, 0.4003906250, 0.4003906250,
    0.3496093750, 0.5000000000, 0.5000000000, 0.5000000000,
    0.5996093750, 0.5996093750, 0.5996093750, 0.5996093750,
    0.5996093750, 0.5996093750, 0.7001953125, 0.7500000000,
    0.7998046875, 0.7998046875
   },
  // non_predictive frame
  {
    0.0000000000, 0.9248046875, 0.9501953125, 0.7001953125,
    0.7001953125, 0.5996093750, 0.5000000000, 0.5000000000,
    0.4501953125, 0.5996093750, 0.6250000000, 0.6250000000,
    0.7753906250, 0.7998046875, 0.7998046875, 0.7998046875,
    0.7998046875, 0.8251953125, 0.9248046875, 1.0000000000,
    1.0000000000, 1.0000000000
  },
};
```

10.4.5 LOGNORM_REFINEMENT_BOOST_TABLE

Table 10-11: Lognorm Refinement Boost Table

Syntax

```
LNFP LOGNORM_REFINEMENT_BOOST_TABLE[NUM_ACE_BANDS] =
{
  0.375000, 0.375000, 0.375000, 0.375000, 0.375000, 0.375000, 0.375000, 0.375000, 0.500000,
  0.500000, 0.500000, 0.625000, 0.625000, 0.625000, 0.625000, 0.679688, 0.734375, 0.789063,
  0.843750, 0.898438, 0.937500, 0.937500
}
```

10.5 Stereo Constants and Tables

10.5.1 Stereo Coding Modes

Table 10-12: Stereo Coding Modes

Syntax

```
// Stereo coding modes

const uint kStereoMono = 0;
const uint kStereoMidSideMono = 1;
const uint kStereoLeftRight = 2;
const uint kStereoMidSide = 3;
```

10.5.2 Stereo Allocation Modes

Table 10-13: Stereo Allocation Modes

```
// Stereo allocation modes

const uint kLrStereoAllocation5050 = 0;        // 50/50 bit distribution
const uint kLrStereoAllocation6040 = 1;        // 60/40 bit distribution
const uint kLrStereoAllocation4060 = 2;        // 40/60 bit distribution
const uint kLrStereoAllocation7525 = 3;        // 75/25 bit distribution
const uint kLrStereoAllocation2575 = 4;        // 25/75 bit distribution
```

10.6 Bit Allocation Constants and Tables

10.6.1 Bit Allocation Models

Table 10-14: Bit Allocation Models

```
// bit allocations are incrementally adjusted from values derived
// from the primary lognorms, with a standard interpretation
// of adjustment values.
const uint kUnsignedDeltaFromPrimaryLSQ = 0;

// bit allocations are derived as (non-negative)
// modifications to a bit allocation model
const uint kUnsignedDeltaFromBasic = 1;

// bit allocations are incrementally adjusted
// from values derived from the primary lognorms, with a special
// interpretation of adjustment values
const uint kUnsignedModifiedDeltaFromPrimaryLSQ = 2;

// bit allocations are adjusted from values derived from the primary
// lognorms, with a standard interpretation of adjustment values
const uint kSignedDeltaFromPrimaryLSQ = 3;

// bit allocations are adjusted from values in a previous frame
const uint kSignedDeltaFromPreviousFrame = 4;

// bit allocation are differentially encoded with respect
// to zero or a bit allocation model
const uint kSignedDiffFromFoundation = 5;
```

10.7 Vector Quantizer Constants and Tables

10.7.1 Vector Quantizer Basic Constants

Table 10-15: Stereo Allocation Modes

```
// VQ constants

cont uint MAX_BAND_SIZE = 208;
```

10.7.2 MAX_BITS_FOR_N

Table 10-16: Max Bits for N

Syntax

```
// entries for N=0 and N=1 are omitted; first entry is for N=2.
const uint MAX_BITS_FOR_N[MAX_BAND_SIZE - 1] =
{
  11, 19, 26, 33, 38, 39, 40, 40,
  41, 41, 41, 41, 42, 42, 42, 42,
  41, 41, 42, 41, 41, 41, 41, 41,
  40, 41, 41, 40, 41, 41, 41, 39,
  39, 40, 40, 41, 39, 39, 39, 39,
  40, 40, 40, 37, 37, 37, 38, 38,
  39, 39, 40, 40, 40, 40, 40, 37,
  37, 37, 37, 37, 37, 38, 38, 38,
  38, 38, 38, 38, 39, 39, 39, 39,
  39, 39, 39, 39, 35, 35, 35, 35,
  35, 36, 36, 36, 36, 36, 36, 36,
  36, 36, 36, 36, 37, 37, 37, 37,
  37, 37, 37, 37, 37, 37, 37, 37,
  38, 38, 38, 38, 38, 38, 38, 38,
  38, 38, 38, 38, 38, 33, 33, 33,
  33, 33, 33, 34, 34, 34, 34, 34,
  34, 34, 34, 34, 34, 34, 34, 34,
  34, 34, 34, 34, 34, 34, 35, 35,
  35, 35, 35, 35, 35, 35, 35, 35,
  35, 35, 35, 35, 35, 35, 35, 35,
  35, 35, 35, 36, 36, 36, 36, 36,
  36, 36, 36, 36, 36, 36, 36, 36,
  36, 36, 36, 36, 36, 36, 36, 36,
  36, 36, 36, 36, 37, 37, 37, 37,
  37, 37, 37, 37, 37, 37, 37, 37,
  37, 37, 37, 37, 37, 37, 37
};
```

10.7.3 MAX_K_FOR_N

Table 10-17: Max K for N

Syntax

```
// Maximum value of K such that pyramid(N,K) fits in 32 bit.

const uint MAX_K_FOR_N[MAX_BAND_SIZE] =
{
  256, 256, 256, 256, 256, 217, 117,  76, 55,
   43,  36,  31,  27,  24,  22,  20,  19, 17,
   16,  16,  15,  14,  14,  13,  13,  12, 12,
   12,  11,  11,  11,  11,  10,  10,  10, 10,
   10,   9,   9,   9,   9,   9,   9,   9,  8,
    8,   8,   8,   8,   8,   8,   8,   8,  8,
    8,   8,   7,   7,   7,   7,   7,   7,  7,
    7,   7,   7,   7,   7,   7,   7,   7,  7,
    7,   7,   7,   7,   7,   6,   6,   6,  6,
    6,   6,   6,   6,   6,   6,   6,   6,  6,
    6,   6,   6,   6,   6,   6,   6,   6,  6,
    6,   6,   6,   6,   6,   6,   6,   6,  6,
    6,   6,   6,   6,   6,   6,   6,   6,  6,
    6,   5,   5,   5,   5,   5,   5,   5,  5,
    5,   5,   5,   5,   5,   5,   5,   5,  5,
    5,   5,   5,   5,   5,   5,   5,   5,  5,
    5,   5,   5,   5,   5,   5,   5,   5,  5,
    5,   5,   5,   5,   5,   5,   5,   5,  5,
    5,   5,   5,   5,   5,   5,   5,   5,  5,
    5,   5,   5,   5,   5,   5,   5,   5,  5,
    5,   5,   5,   5,   5,   5,   5,   5,  5,
    5,   5,   5,   5,   5,   5,   5,   5,  5,
    5,   5,   5,   5,   5,   5,   5,   5,  5,
    5
};
```

10.7.4 K_ESTIMATION_COEFFICIENTS

Table 10-18: K Estimation Coefficients

Syntax

```
// K-estimation coefficients.

const uint K_ESTIMATION_COEFFICIENTS[] = {
  {411, 128}, {733, 255}, {671, 367}, {726, 471},
  {597, 571}, {243, 656}, {118, 751}, {520, 887},
  {679, 1007}, {831, 1121}, {1044, 1251}, {1166, 1370},
  {1329, 1493}, {1458, 1623}, {1637, 1762}, {1761, 1861},
  {1973, 2012}, {2205, 2187}, {2297, 2292}, {2331, 2391},
  {2584, 2557}, {2658, 2647}, {2809, 2792}, {2956, 2915},
  {3121, 3063}, {3147, 3125}, {3234, 3225}, {3523, 3441},
  {3526, 3484}, {3691, 3638}, {3762, 3714}, {3949, 3862},
  {4186, 4060}, {4207, 4111}, {4357, 4241}, {4477, 4366},
  {4511, 4424}, {4731, 4623}, {4693, 4626}, {4894, 4785},
  {4946, 4860}, {5011, 4942}, {5104, 5049}, {5407, 5293},
  {5380, 5283}, {5384, 5303}, {5586, 5492}, {5724, 5620},
  {5904, 5791}, {6039, 5928}, {5851, 5801}, {6103, 6022},
  {6147, 6082}, {6267, 6207}, {6465, 6389}, {6502, 6436},
  {6454, 6410}, {6639, 6578}, {6794, 6712}, {6804, 6756},
  {6853, 6808}, {6966, 6923}, {7315, 7214}, {7701, 7516},
  {7578, 7451}, {7777, 7639}, {7913, 7778}, {7965, 7834},
  {7965, 7854}, {8034, 7925}, {8185, 8055}, {8377, 8232},
  {8354, 8222}, {8403, 8290}, {8503, 8394}, {8746, 8619},
  {8963, 8818}, {8844, 8735}, {8976, 8875}, {8976, 8875},
  {9099, 9000}, {8857, 8792}, {9099, 9021}, {9254, 9165},
  {9469, 9357}, {9200, 9165}, {9485, 9420}, {9604, 9532},
  {9820, 9736}, {10104, 9983}, {10451, 10284}, {10317, 10179},
  {10083, 9985}, {10139, 10057}, {10139, 10057}, {10506, 10393},
  {10339, 10273}, {10547, 10457}, {10696, 10597}, {10962, 10853},
  {10962, 10853}, {11011, 10921}, {11166, 11079}, {11164, 11089},
  {11327, 11243}, {11638, 11526}, {11460, 11382}, {11139, 11101},
  {11622, 11548}, {11622, 11548}, {11622, 11548}, {12181, 12067},
  {11967, 11904}, {11605, 11597}, {12047, 11991}, {12206, 12144},
  {12047, 11991}, {11777, 11754}, {11777, 11754}, {12285, 12227},
  {12068, 12048}, {12068, 12048}, {12267, 12243}, {12840, 12778},
  {12840, 12778}, {12840, 12778}, {13045, 12947}, {13766, 13580},
  {13686, 13580}, {13686, 13580}, {13686, 13580}, {13686, 13580},
  {14588, 14424}, {14588, 14424}, {14871, 14701}, {14871, 14701},
  {14604, 14477}, {15388, 15212}, {15388, 15212}, {14604, 14477},
  {14835, 14714}, {14835, 14714}, {15753, 15573}, {15753, 15573},
  {14503, 14424}, {14503, 14424}, {14214, 14178}, {14596, 14555},
  {15412, 15323}, {15412, 15323}, {15412, 15323}, {15412, 15323},
  {15412, 15323}, {15412, 15323}, {16622, 16478}, {16622, 16478},
  {16622, 16478}, {16239, 16147}, {16239, 16147}, {16239, 16147},
  {16239, 16147}, {17216, 17071}, {17216, 17071}, {17613, 17461},
  {16525, 16478}, {15570, 15573}, {16525, 16478}, {16525, 16478},
  {16525, 16478}, {16525, 16478}, {16145, 16147}, {16145, 16147},
  {16145, 16147}, {16393, 16400}, {17511, 17461}, {17511, 17461},
  {17511, 17461}, {17511, 17461}, {17511, 17461}, {17861, 17762},
  {17861, 17762}, {17511, 17461}, {18687, 18579}, {18687, 18579},
  {18687, 18579}, {18687, 18579}, {18687, 18579}, {18687, 18579},
  {18687, 18579}, {18687, 18579}, {18687, 18579}, {18687, 18579},
  {19324, 19260}, {19324, 19260}, {19324, 19260}, {19324, 19260},
  {19324, 19260}, {17408, 17461}, {18976, 18960}, {18976, 18960},
  {18976, 18960}, {18976, 18960}, {18976, 18960}, {18976, 18960},
  {19403, 19386}, {19403, 19386}, {19403, 19386}};
```

10.7.5 BETA_DEQUANT_TABLE

Table 10-19: Beta Dequantization Table

Syntax

```
const uint MAX_QUANT_LEVEL = 256;

int BETA_DEQUANT_TABLE[MAX_QUANT_LEVEL/2][MAX_QUANT_LEVEL+1] =
// The entry corresponding to a given pair (quantization_level, index) is found
// in BETA_DEQUANT_TABLE[quantization_level/2-1][index] if
// index <= quantization_level/2; otherwise, it is obtained as
// -BETA_DEQUANT_TABLE[quantization_level/2-1][quantization_level/2-index].
```

Syntax

```
// Assumption: quantization_level is always even.
{
  { // quantization_level=  2
    -32767,      0
  },
  { // quantization_level=  4
    -32767,  -2611,      0
  },
  { // quantization_level=  6
    -32767,  -3862,  -1659,      0
  },
  { // quantization_level=  8
    -32767,  -4769,  -2611,  -1194,      0
  },
  { // quantization_level= 10
    -32767,  -5397,  -3311,  -2035,   -951,      0
  },
  { // quantization_level= 12
    -32767,  -5954,  -3862,  -2611,  -1659,   -783,      0
  },
  { // quantization_level= 14
    -32767,  -6398,  -4319,  -3104,  -2190,  -1394,   -671,      0
  },
  { // quantization_level= 16
    -32767,  -6839,  -4769,  -3528,  -2611,  -1903,  -1194,   -597,
     0
  },
  { // quantization_level= 18
    -32767,  -7125,  -5121,  -3862,  -2990,  -2279,  -1659,  -1044,
     -522,      0
  },
  { // quantization_level= 20
    -32767,  -7494,  -5397,  -4196,  -3311,  -2611,  -2035,  -1460,
     -951,   -466,      0
  },
  { // quantization_level= 22
    -32767,  -7746,  -5693,  -4490,  -3591,  -2906,  -2345,  -1836,
    -1305,   -858,   -429,      0
  },
  { // quantization_level= 24
    -32767,  -8006,  -5954,  -4769,  -3862,  -3191,  -2611,  -2124,
    -1659,  -1194,   -783,   -391,      0
  },
  { // quantization_level= 26
    -32767,  -8278,  -6166,  -4965,  -4120,  -3433,  -2851,  -2389,
    -1947,  -1504,  -1100,   -727,   -354,      0
  },
  { // quantization_level= 28
    -32767,  -8419,  -6398,  -5229,  -4319,  -3657,  -3104,  -2611,
    -2190,  -1792,  -1394,  -1007,   -671,   -335,      0
  },
  { // quantization_level= 30
    -32767,  -8568,  -6566,  -5397,  -4534,  -3862,  -3311,  -2824,
    -2411,  -2035,  -1659,  -1283,   -951,   -634,   -317,      0
  },
  { // quantization_level= 32
    -32767,  -8886,  -6839,  -5632,  -4769,  -4082,  -3528,  -3047,
    -2611,  -2257,  -1903,  -1549,  -1194,   -895,   -597,   -298, 0
  },
  { // quantization_level= 34
    -32767,  -8886,  -6883,  -5756,  -4913,  -4237,  -3690,  -3220,
    -2796,  -2434,  -2102,  -1770,  -1438,  -1119,   -839,   -559,
     -279,      0
  },
  { // quantization_level= 36
    -32767,  -9061,  -7125,  -5954,  -5121,  -4402,  -3862,  -3402,
    -2990,  -2611,  -2279,  -1969,  -1659,  -1349,  -1044,   -783,
     -522,   -261,      0
  },
  { // quantization_level= 38
    -32767,  -9249,  -7370,  -6094,  -5284,  -4581,  -4043,  -3559,
    -3163,  -2769,  -2456,  -2146,  -1858,  -1549,  -1261,   -988,
     -746,   -485,   -242,      0
  },
  { // quantization_level= 40
    -32767,  -9452,  -7494,  -6242,  -5397,  -4769,  -4196,  -3723,
    -3311,  -2935,  -2611,  -2323,  -2035,  -1748,  -1460,  -1194,
```

Syntax

```
    -951,   -708,   -466,   -223,      0
  },
  { // quantization_level= 42
    -32767,  -9452,  -7619,  -6398,  -5571,  -4913,  -4319,  -3862,
     -3464,  -3104,  -2769,  -2456,  -2190,  -1925,  -1659,  -1394,
     -1119,   -895,   -671,   -447,   -223,      0
  },
  { // quantization_level= 44
    -32767,  -9678,  -7746,  -6566,  -5693,  -5017,  -4490,  -4006,
     -3591,  -3250,  -2906,  -2611,  -2345,  -2080,  -1836,  -1571,
     -1305,  -1082,   -858,   -634,   -429,   -205,      0
  },
  { // quantization_level= 46
    -32767,  -9678,  -7874,  -6652,  -5820,  -5175,  -4627,  -4157,
     -3723,  -3370,  -3047,  -2742,  -2478,  -2234,  -1991,  -1748,
     -1482,  -1239,  -1026,   -820,   -615,   -410,   -205,      0
  },
  { // quantization_level= 48
    -32767,  -9926,  -8006,  -6839,  -5954,  -5284,  -4769,  -4278,
     -3862,  -3528,  -3191,  -2879,  -2611,  -2367,  -2124,  -1903,
     -1659,  -1416,  -1194,   -988,   -783,   -597,   -391,   -186,
     0
  },
  { // quantization_level= 50
    -32767,  -9926,  -8139,  -6883,  -6094,  -5397,  -4862,  -4402,
     -4006,  -3623,  -3311,  -3018,  -2742,  -2478,  -2257,  -2035,
     -1814,  -1571,  -1349,  -1138,   -951,   -746,   -559,   -373,
      -186,      0
  },
  { // quantization_level= 52
    -32767, -10206,  -8278,  -7004,  -6166,  -5512,  -4965,  -4534,
     -4120,  -3757,  -3433,  -3133,  -2851,  -2611,  -2389,  -2168,
     -1947,  -1726,  -1504,  -1283,  -1100,   -914,   -727,   -541,
      -354,   -167,      0
  },
  { // quantization_level= 54
    -32767, -10206,  -8419,  -7125,  -6317,  -5632,  -5121,  -4627,
     -4237,  -3862,  -3559,  -3250,  -2990,  -2715,  -2500,  -2279,
     -2080,  -1858,  -1659,  -1438,  -1239,  -1044,   -876,   -690,
      -522,   -335,   -167,      0
  },
  { // quantization_level= 56
    -32767, -10206,  -8419,  -7248,  -6398,  -5756,  -5229,  -4769,
     -4319,  -3969,  -3657,  -3370,  -3104,  -2851,  -2611,  -2389,
     -2190,  -1991,  -1792,  -1593,  -1394,  -1194,  -1007,   -839,
      -671,   -503,   -335,   -167,      0
  },
  { // quantization_level= 58
    -32767, -10527,  -8568,  -7370,  -6479,  -5820,  -5340,  -4862,
     -4446,  -4082,  -3757,  -3464,  -3220,  -2963,  -2715,  -2500,
     -2301,  -2102,  -1925,  -1726,  -1526,  -1327,  -1138,   -970,
      -820,   -652,   -485,   -317,   -149,      0
  },
  { // quantization_level= 60
    -32767, -10527,  -8568,  -7494,  -6566,  -5954,  -5397,  -4965,
     -4534,  -4196,  -3862,  -3591,  -3311,  -3076,  -2824,  -2611,
     -2411,  -2212,  -2035,  -1836,  -1659,  -1460,  -1283,  -1100,
      -951,   -783,   -634,   -466,   -317,   -149,      0
  },
  { // quantization_level= 62
    -32767, -10527,  -8723,  -7619,  -6652,  -6022,  -5512,  -5069,
     -4627,  -4278,  -3969,  -3690,  -3402,  -3163,  -2935,  -2715,
     -2500,  -2323,  -2146,  -1969,  -1770,  -1593,  -1416,  -1239,
     -1063,   -914,   -764,   -615,   -447,   -298,   -149,      0
  },
  { // quantization_level= 64
    -32767, -10892,  -8886,  -7746,  -6839,  -6166,  -5632,  -5175,
     -4769,  -4402,  -4082,  -3792,  -3528,  -3281,  -3047,  -2824,
     -2611,  -2434,  -2257,  -2080,  -1903,  -1726,  -1549,  -1372,
     -1194,  -1044,   -895,   -746,   -597,   -447,   -298,   -149, 0
  },
  { // quantization_level= 66
    -32767, -10892,  -8886,  -7746,  -6763,  -6242,  -5693,  -5229,
     -4812,  -4490,  -4157,  -3862,  -3591,  -3370,  -3133,  -2906,
     -2688,  -2522,  -2345,  -2168,  -1991,  -1836,  -1659,  -1482,
     -1305,  -1156,  -1007,   -858,   -708,   -578,   -429,   -279,
      -130,      0
```

Syntax

```
},
{ // quantization_level= 68
  -32767, -10892,  -8886,  -7874,  -6883,  -6317,  -5756,  -5340,
   -4913,  -4581,  -4237,  -3969,  -3690,  -3464,  -3220,  -3018,
   -2796,  -2611,  -2434,  -2257,  -2102,  -1925,  -1770,  -1593,
   -1438,  -1261,  -1119,   -970,   -839,   -690,   -559,   -410,
    -279,   -130,      0
},
{ // quantization_level= 70
  -32767, -10892,  -9061,  -8006,  -7004,  -6398,  -5885,  -5397,
   -5017,  -4671,  -4319,  -4043,  -3792,  -3528,  -3311,  -3104,
   -2879,  -2688,  -2522,  -2367,  -2190,  -2035,  -1880,  -1703,
   -1549,  -1394,  -1217,  -1082,   -951,   -802,   -671,   -541,
    -391,   -261,   -130,      0
},
{ // quantization_level= 72
  -32767, -10892,  -9061,  -8006,  -7125,  -6479,  -5954,  -5512,
   -5121,  -4769,  -4402,  -4120,  -3862,  -3623,  -3402,  -3191,
   -2990,  -2796,  -2611,  -2434,  -2279,  -2124,  -1969,  -1814,
   -1659,  -1504,  -1349,  -1194,  -1044,   -914,   -783,   -652,
    -522,   -391,   -261,   -130,      0
},
{ // quantization_level= 74
  -32767, -11324,  -9249,  -8139,  -7248,  -6566,  -6022,  -5571,
   -5175,  -4812,  -4490,  -4196,  -3932,  -3723,  -3495,  -3281,
   -3076,  -2879,  -2688,  -2522,  -2367,  -2212,  -2057,  -1903,
   -1748,  -1615,  -1460,  -1305,  -1156,  -1026,   -895,   -764,
    -634,   -503,   -373,   -242,   -111,      0
},
{ // quantization_level= 76
  -32767, -11324,  -9249,  -8139,  -7370,  -6652,  -6094,  -5632,
   -5284,  -4913,  -4581,  -4278,  -4043,  -3792,  -3559,  -3341,
   -3163,  -2963,  -2769,  -2611,  -2456,  -2301,  -2146,  -2013,
   -1858,  -1703,  -1549,  -1416,  -1261,  -1119,   -988,   -876,
    -746,   -615,   -485,   -373,   -242,   -111,      0
},
{ // quantization_level= 78
  -32767, -11324,  -9249,  -8278,  -7370,  -6745,  -6166,  -5756,
   -5340,  -4965,  -4671,  -4361,  -4120,  -3862,  -3657,  -3433,
   -3220,  -3047,  -2851,  -2688,  -2522,  -2389,  -2234,  -2102,
   -1947,  -1792,  -1659,  -1504,  -1372,  -1217,  -1100,   -970,
    -839,   -727,   -597,   -485,   -354,   -242,   -111,      0
},
{ // quantization_level= 80
  -32767, -11324,  -9452,  -8278,  -7494,  -6839,  -6242,  -5820,
   -5397,  -5069,  -4769,  -4446,  -4196,  -3932,  -3723,  -3528,
   -3311,  -3133,  -2935,  -2769,  -2611,  -2456,  -2323,  -2168,
   -2035,  -1903,  -1748,  -1615,  -1460,  -1327,  -1194,  -1063,
    -951,   -820,   -708,   -597,   -466,   -354,   -223,   -111,
   0
},
{ // quantization_level= 82
  -32767, -11324,  -9452,  -8419,  -7619,  -6763,  -6317,  -5885,
   -5512,  -5121,  -4812,  -4534,  -4278,  -4006,  -3792,  -3591,
   -3402,  -3191,  -3018,  -2851,  -2688,  -2522,  -2389,  -2257,
   -2124,  -1969,  -1836,  -1703,  -1571,  -1416,  -1283,  -1156,
   -1044,   -914,   -802,   -690,   -578,   -447,   -335,   -223,
    -111,      0
},
{ // quantization_level= 84
  -32767, -11324,  -9452,  -8419,  -7619,  -6883,  -6398,  -5954,
   -5571,  -5229,  -4913,  -4581,  -4319,  -4082,  -3862,  -3657,
   -3464,  -3281,  -3104,  -2935,  -2769,  -2611,  -2456,  -2323,
   -2190,  -2057,  -1925,  -1792,  -1659,  -1526,  -1394,  -1261,
   -1119,  -1007,   -895,   -783,   -671,   -559,   -447,   -335,
    -223,   -111,      0
},
{ // quantization_level= 86
  -32767, -11819,  -9678,  -8568,  -7746,  -7004,  -6479,  -6022,
   -5632,  -5284,  -4965,  -4671,  -4402,  -4157,  -3932,  -3723,
   -3528,  -3341,  -3163,  -2990,  -2824,  -2661,  -2544,  -2411,
   -2279,  -2146,  -2013,  -1880,  -1748,  -1615,  -1482,  -1349,
   -1217,  -1100,   -988,   -876,   -764,   -652,   -541,   -429,
    -317,   -205,    -93,      0
},
{ // quantization_level= 88
  -32767, -11819,  -9678,  -8568,  -7746,  -7004,  -6566,  -6094,
```

Syntax

```
    -5693,  -5340,  -5017,  -4769,  -4490,  -4237,  -4006,  -3792,
    -3591,  -3433,  -3250,  -3076,  -2906,  -2742,  -2611,  -2478,
    -2345,  -2212,  -2080,  -1947,  -1836,  -1703,  -1571,  -1438,
    -1305,  -1194,  -1082,   -970,   -858,   -746,   -634,   -541,
     -429,   -317,   -205,    -93,      0
  },
  { // quantization_level= 90
    -32767, -11819,  -9678,  -8568,  -7874,  -7125,  -6566,  -6166,
     -5756,  -5397,  -5121,  -4812,  -4534,  -4319,  -4082,  -3862,
     -3657,  -3495,  -3311,  -3133,  -2990,  -2824,  -2661,  -2544,
     -2411,  -2279,  -2168,  -2035,  -1903,  -1792,  -1659,  -1526,
     -1394,  -1283,  -1156,  -1044,   -951,   -839,   -727,   -634,
      -522,   -410,   -317,   -205,    -93,      0
  },
  { // quantization_level= 92
    -32767, -11819,  -9678,  -8723,  -7874,  -7248,  -6652,  -6242,
     -5820,  -5453,  -5175,  -4862,  -4627,  -4361,  -4157,  -3932,
     -3723,  -3559,  -3370,  -3220,  -3047,  -2906,  -2742,  -2611,
     -2478,  -2345,  -2234,  -2102,  -1991,  -1858,  -1748,  -1615,
     -1482,  -1372,  -1239,  -1138,  -1026,   -932,   -820,   -708,
      -615,   -503,   -410,   -298,   -205,    -93,      0
  },
  { // quantization_level= 94
    -32767, -11819,  -9926,  -8723,  -8006,  -7248,  -6745,  -6242,
     -5885,  -5512,  -5229,  -4965,  -4671,  -4446,  -4196,  -4006,
     -3792,  -3623,  -3433,  -3281,  -3133,  -2963,  -2824,  -2661,
     -2544,  -2411,  -2301,  -2168,  -2057,  -1947,  -1814,  -1703,
     -1571,  -1460,  -1327,  -1217,  -1100,  -1007,   -914,   -802,
      -708,   -597,   -503,   -391,   -298,   -186,    -93,      0
  },
  { // quantization_level= 96
    -32767, -11819,  -9926,  -8886,  -8006,  -7370,  -6839,  -6317,
     -5954,  -5632,  -5284,  -5017,  -4769,  -4490,  -4278,  -4082,
     -3862,  -3690,  -3528,  -3341,  -3191,  -3047,  -2879,  -2742,
     -2611,  -2478,  -2367,  -2257,  -2124,  -2013,  -1903,  -1770,
     -1659,  -1549,  -1416,  -1305,  -1194,  -1082,   -988,   -895,
      -783,   -690,   -597,   -485,   -391,   -298,   -186,    -93,
     0
  },
  { // quantization_level= 98
    -32767, -11819,  -9926,  -8886,  -8139,  -7370,  -6763,  -6398,
     -6022,  -5632,  -5340,  -5069,  -4812,  -4581,  -4319,  -4120,
     -3932,  -3757,  -3559,  -3402,  -3250,  -3104,  -2963,  -2796,
     -2661,  -2544,  -2434,  -2301,  -2190,  -2080,  -1969,  -1858,
     -1726,  -1615,  -1504,  -1394,  -1261,  -1156,  -1063,   -970,
      -876,   -764,   -671,   -578,   -485,   -373,   -279,   -186,
       -93,      0
  },
  { // quantization_level=100
    -32767, -11819,  -9926,  -8886,  -8139,  -7494,  -6883,  -6479,
     -6094,  -5693,  -5397,  -5121,  -4862,  -4627,  -4402,  -4196,
     -4006,  -3792,  -3623,  -3464,  -3311,  -3163,  -3018,  -2879,
     -2742,  -2611,  -2478,  -2367,  -2257,  -2146,  -2035,  -1925,
     -1814,  -1703,  -1571,  -1460,  -1349,  -1239,  -1138,  -1044,
      -951,   -858,   -746,   -652,   -559,   -466,   -373,   -279,
      -186,    -93,      0
  },
  { // quantization_level=102
    -32767, -11819,  -9926,  -8886,  -8139,  -7494,  -6883,  -6479,
     -6094,  -5756,  -5453,  -5175,  -4913,  -4671,  -4446,  -4237,
     -4043,  -3862,  -3690,  -3528,  -3370,  -3220,  -3076,  -2935,
     -2796,  -2661,  -2544,  -2434,  -2323,  -2212,  -2102,  -1991,
     -1880,  -1770,  -1659,  -1549,  -1438,  -1327,  -1217,  -1119,
     -1026,   -932,   -839,   -746,   -652,   -559,   -466,   -373,
      -279,   -186,    -93,      0
  },
  { // quantization_level=104
    -32767, -12403, -10206,  -9061,  -8278,  -7619,  -7004,  -6566,
     -6166,  -5820,  -5512,  -5229,  -4965,  -4769,  -4534,  -4319,
     -4120,  -3932,  -3757,  -3591,  -3433,  -3281,  -3133,  -2990,
     -2851,  -2715,  -2611,  -2500,  -2389,  -2279,  -2168,  -2057,
     -1947,  -1836,  -1726,  -1615,  -1504,  -1394,  -1283,  -1194,
     -1100,  -1007,   -914,   -820,   -727,   -634,   -541,   -447,
      -354,   -261,   -167,    -74,      0
  },
  { // quantization_level=106
    -32767, -12403, -10206,  -9061,  -8278,  -7619,  -7125,  -6652,
```

Syntax

```
    -6242,  -5885,  -5571,  -5284,  -5069,  -4812,  -4581,  -4361,
    -4157,  -3969,  -3827,  -3657,  -3495,  -3341,  -3191,  -3047,
    -2935,  -2796,  -2661,  -2544,  -2434,  -2323,  -2234,  -2124,
    -2013,  -1903,  -1792,  -1681,  -1593,  -1482,  -1372,  -1261,
    -1156,  -1063,   -988,   -895,   -802,   -708,   -615,   -522,
     -447,   -354,   -261,   -167,    -74,      0
  },
  { // quantization_level=108
    -32767, -12403, -10206,  -9061,  -8419,  -7746,  -7125,  -6652,
    -6317,  -5954,  -5632,  -5340,  -5121,  -4862,  -4627,  -4402,
    -4237,  -4043,  -3862,  -3690,  -3559,  -3402,  -3250,  -3104,
    -2990,  -2851,  -2715,  -2611,  -2500,  -2389,  -2279,  -2190,
    -2080,  -1969,  -1858,  -1770,  -1659,  -1549,  -1438,  -1349,
    -1239,  -1138,  -1044,   -970,   -876,   -783,   -690,   -615,
     -522,   -429,   -335,   -261,   -167,    -74,      0
  },
  { // quantization_level=110
    -32767, -12403, -10206,  -9249,  -8419,  -7746,  -7248,  -6745,
    -6317,  -6022,  -5693,  -5397,  -5175,  -4913,  -4671,  -4490,
    -4278,  -4082,  -3932,  -3757,  -3591,  -3464,  -3311,  -3163,
    -3047,  -2906,  -2769,  -2661,  -2544,  -2456,  -2345,  -2234,
    -2146,  -2035,  -1925,  -1836,  -1726,  -1615,  -1526,  -1416,
    -1305,  -1217,  -1119,  -1026,   -951,   -858,   -764,   -690,
     -597,   -503,   -429,   -335,   -242,   -167,    -74,      0
  },
  { // quantization_level=112
    -32767, -12403, -10206, -9249,  -8419,  -7874,  -7248,  -6839,
    -6398,  -6022,  -5756,  -5453,  -5229,  -4965,  -4769,  -4534,
    -4319,  -4157,  -3969,  -3827,  -3657,  -3528,  -3370,  -3220,
    -3104,  -2963,  -2851,  -2715,  -2611,  -2500,  -2389,  -2301,
    -2190,  -2102,  -1991,  -1903,  -1792,  -1681,  -1593,  -1482,
    -1394,  -1283,  -1194,  -1100,  -1007,  -932,   -839,   -764,
    -671,   -597,   -503,   -410,   -335,   -242,   -167,   -74,
     0
  },
  { // quantization_level=114
    -32767, -12403, -10527,  -9249,  -8568,  -7874,  -7370,  -6763,
     -6479,  -6094,  -5820,  -5512,  -5284,  -5017,  -4812,  -4581,
     -4402,  -4196,  -4043,  -3862,  -3723,  -3559,  -3433,  -3281,
     -3163,  -3018,  -2906,  -2769,  -2661,  -2544,  -2456,  -2345,
     -2257,  -2146,  -2057,  -1947,  -1858,  -1748,  -1659,  -1549,
     -1460,  -1349,  -1261,  -1156,  -1082,   -988,   -914,   -820,
      -746,   -652,   -578,   -485,   -410,   -317,   -242,   -149,
       -74,      0
  },
  { // quantization_level=116
    -32767, -12403, -10527,  -9249,  -8568,  -7874,  -7370,  -6883,
     -6479,  -6166,  -5820,  -5571,  -5340,  -5069,  -4862,  -4627,
     -4446,  -4237,  -4082,  -3932,  -3757,  -3623,  -3464,  -3341,
     -3220,  -3076,  -2963,  -2824,  -2715,  -2611,  -2500,  -2411,
     -2301,  -2212,  -2102,  -2013,  -1925,  -1814,  -1726,  -1615,
     -1526,  -1438,  -1327,  -1239,  -1138,  -1063,   -970,   -895,
      -820,   -727,   -652,   -559,   -485,   -410,   -317,   -242,
      -149,    -74,      0
  },
  { // quantization_level=118
    -32767, -12403, -10527,  -9249,  -8568,  -8006,  -7370,  -6883,
     -6566,  -6166,  -5885,  -5632,  -5340,  -5121,  -4913,  -4671,
     -4490,  -4319,  -4120,  -3969,  -3827,  -3657,  -3528,  -3402,
     -3250,  -3133,  -3018,  -2879,  -2769,  -2661,  -2544,  -2456,
     -2367,  -2257,  -2168,  -2080,  -1969,  -1880,  -1792,  -1681,
     -1593,  -1504,  -1394,  -1305,  -1217,  -1119,  -1044,   -970,
      -876,   -802,   -727,   -634,   -559,   -485,   -391,   -317,
      -242,   -149,    -74,      0
  },
  { // quantization_level=120
    -32767, -12403, -10527,  -9452,  -8568,  -8006,  -7494,  -7004,
     -6566,  -6242,  -5954,  -5693,  -5397,  -5175,  -4965,  -4769,
     -4534,  -4361,  -4196,  -4006,  -3862,  -3723,  -3591,  -3433,
     -3311,  -3191,  -3076,  -2935,  -2824,  -2715,  -2611,  -2500,
     -2411,  -2323,  -2212,  -2124,  -2035,  -1947,  -1836,  -1748,
     -1659,  -1571,  -1460,  -1372,  -1283,  -1194,  -1100,  -1026,
      -951,   -858,   -783,   -708,   -634,   -541,   -466,   -391,
      -317,   -223,   -149,    -74,      0
  },
  { // quantization_level=122
    -32767, -12403, -10527,  -9452,  -8723,  -8139,  -7494,  -7004,
```

Syntax

```
    -6652,  -6317,  -6022,  -5693,  -5453,  -5229,  -5017,  -4812,
    -4581,  -4402,  -4237,  -4082,  -3932,  -3757,  -3623,  -3495,
    -3370,  -3250,  -3104,  -2990,  -2879,  -2769,  -2661,  -2544,
    -2456,  -2367,  -2279,  -2190,  -2080,  -1991,  -1903,  -1814,
    -1726,  -1615,  -1526,  -1438,  -1349,  -1261,  -1156,  -1082,
    -1007,   -932,   -858,   -764,   -690,   -615,   -541,   -466,
     -373,   -298,   -223,   -149,    -74,      0
  },
  { // quantization_level=124
    -32767, -12403, -10527,  -9452,  -8723,  -8139,  -7619,  -7125,
    -6652,  -6317,  -6022,  -5756,  -5512,  -5284,  -5069,  -4862,
    -4627,  -4446,  -4278,  -4120,  -3969,  -3827,  -3690,  -3559,
    -3402,  -3281,  -3163,  -3047,  -2935,  -2824,  -2715,  -2611,
    -2500,  -2411,  -2323,  -2234,  -2146,  -2057,  -1969,  -1858,
    -1770,  -1681,  -1593,  -1504,  -1416,  -1327,  -1239,  -1138,
    -1063,   -988,   -914,   -839,   -764,   -690,   -615,   -522,
     -447,   -373,   -298,   -223,   -149,    -74,      0
  },
  { // quantization_level=126
    -32767, -12403, -10527,  -9452,  -8723,  -8139,  -7619,  -7125,
    -6745,  -6398,  -6094,  -5820,  -5571,  -5340,  -5121,  -4913,
    -4671,  -4490,  -4319,  -4157,  -4006,  -3862,  -3723,  -3591,
    -3464,  -3341,  -3220,  -3104,  -2990,  -2879,  -2769,  -2661,
    -2544,  -2456,  -2367,  -2279,  -2190,  -2102,  -2013,  -1925,
    -1836,  -1748,  -1659,  -1571,  -1482,  -1394,  -1305,  -1217,
    -1119,  -1044,   -970,   -895,   -820,   -746,   -671,   -597,
     -522,   -447,   -373,   -298,   -223,   -149,    -74,      0
  },
  { // quantization_level=128
    -32767, -13114, -10892,  -9678,  -8886,  -8278,  -7746,  -7248,
    -6839,  -6479,  -6166,  -5885,  -5632,  -5397,  -5175,  -4965,
    -4769,  -4581,  -4402,  -4237,  -4082,  -3932,  -3792,  -3657,
    -3528,  -3402,  -3281,  -3163,  -3047,  -2935,  -2824,  -2715,
    -2611,  -2522,  -2434,  -2345,  -2257,  -2168,  -2080,  -1991,
    -1903,  -1814,  -1726,  -1637,  -1549,  -1460,  -1372,  -1283,
    -1194,  -1119,  -1044,   -970,   -895,   -820,   -746,   -671,
     -597,   -522,   -447,   -373,   -298,   -223,   -149,    -74,
    0
  },
  { // quantization_level=130
    -32767, -13114, -10892,  -9678,  -8886,  -8278,  -7746,  -7248,
    -6763,  -6479,  -6166,  -5885,  -5632,  -5397,  -5175,  -4965,
    -4762,  -4627,  -4446,  -4278,  -4120,  -3969,  -3827,  -3690,
    -3559,  -3433,  -3311,  -3191,  -3076,  -2963,  -2851,  -2742,
    -2635,  -2566,  -2478,  -2389,  -2301,  -2212,  -2124,  -2035,
    -1947,  -1858,  -1770,  -1681,  -1593,  -1504,  -1416,  -1327,
    -1239,  -1175,  -1100,  -1026,   -951,   -876,   -802,   -727,
     -652,   -578,   -503,   -429,   -354,   -279,   -205,   -130,
      -55,      0
  },
  { // quantization_level=132
    -32767, -13114, -10892,  -9678,  -8886,  -8278,  -7746,  -7248,
    -6763,  -6566,  -6242,  -5954,  -5693,  -5453,  -5229,  -5017,
    -4812,  -4671,  -4490,  -4319,  -4157,  -4006,  -3862,  -3723,
    -3591,  -3495,  -3370,  -3250,  -3133,  -3018,  -2906,  -2796,
    -2688,  -2611,  -2522,  -2434,  -2345,  -2257,  -2168,  -2080,
    -1991,  -1903,  -1836,  -1748,  -1659,  -1571,  -1482,  -1394,
    -1305,  -1217,  -1156,  -1082,  -1007,   -932,   -858,   -783,
     -708,   -634,   -578,   -503,   -429,   -354,   -279,   -205,
     -130,    -55,      0
  },
  { // quantization_level=134
    -32767, -13114, -10892,  -9678,  -8886,  -8278,  -7874,  -7370,
    -6883,  -6566,  -6242,  -5954,  -5756,  -5512,  -5284,  -5069,
    -4862,  -4719,  -4534,  -4361,  -4196,  -4043,  -3897,  -3792,
    -3657,  -3528,  -3402,  -3281,  -3191,  -3076,  -2963,  -2851,
    -2742,  -2635,  -2566,  -2478,  -2389,  -2301,  -2212,  -2124,
    -2057,  -1969,  -1880,  -1792,  -1703,  -1637,  -1549,  -1460,
    -1372,  -1283,  -1195,  -1138,  -1063,   -988,   -914,   -839,
     -783,   -708,   -634,   -559,   -485,   -410,   -354,   -279,
     -205,   -130,    -55,      0
  },
  { // quantization_level=136
    -32767, -13114, -10892,  -9678,  -8886,  -8419,  -7874,  -7370,
    -6883,  -6652,  -6317,  -6022,  -5756,  -5571,  -5340,  -5121,
    -4913,  -4769,  -4581,  -4402,  -4237,  -4082,  -3969,  -3827,
    -3690,  -3559,  -3464,  -3341,  -3220,  -3104,  -3018,  -2906,
```

Syntax

```
    -2796,  -2688,  -2611,  -2522,  -2434,  -2345,  -2257,  -2190,
    -2102,  -2013,  -1925,  -1858,  -1770,  -1681,  -1593,  -1526,
    -1438,  -1349,  -1261,  -1194,  -1119,  -1044,   -970,   -895,
     -839,   -764,   -690,   -615,   -559,   -485,   -410,   -335,
     -279,   -205,   -130,    -55,      0
  },
  { // quantization_level=138
    -32767, -13114, -10892,  -9678,  -9061,  -8419,  -7874,  -7494,
     -7004,  -6652,  -6317,  -6094,  -5820,  -5571,  -5397,  -5175,
     -4965,  -4762,  -4627,  -4446,  -4278,  -4157,  -4006,  -3862,
     -3723,  -3623,  -3495,  -3370,  -3281,  -3163,  -3047,  -2935,
     -2851,  -2742,  -2635,  -2566,  -2478,  -2389,  -2323,  -2234,
     -2146,  -2057,  -1991,  -1903,  -1814,  -1748,  -1659,  -1571,
     -1482,  -1416,  -1327,  -1239,  -1175,  -1100,  -1026,   -951,
      -895,   -820,   -746,   -690,   -615,   -541,   -466,   -410,
      -335,   -261,   -205,   -130,    -55,      0
  },
  { // quantization_level=140
    -32767, -13114, -10892,  -9926,  -9061,  -8419,  -8006,  -7494,
     -7004,  -6745,  -6398,  -6094,  -5885,  -5632,  -5397,  -5229,
     -5017,  -4812,  -4671,  -4490,  -4319,  -4196,  -4043,  -3897,
     -3792,  -3657,  -3528,  -3433,  -3311,  -3191,  -3104,  -2990,
     -2879,  -2796,  -2688,  -2611,  -2522,  -2434,  -2367,  -2279,
     -2190,  -2124,  -2035,  -1947,  -1880,  -1792,  -1703,  -1637,
     -1549,  -1460,  -1394,  -1305,  -1217,  -1156,  -1082,  -1007,
      -951,   -876,   -802,   -746,   -671,   -597,   -541,   -466,
      -391,   -335,   -261,   -186,   -130,    -55,      0
  },
  { // quantization_level=142
    -32767, -13114, -10892,  -9926,  -9061,  -8419,  -8006,  -7494,
     -7125,  -6745,  -6398,  -6166,  -5885,  -5693,  -5453,  -5229,
     -5069,  -4862,  -4719,  -4534,  -4361,  -4237,  -4082,  -3969,
     -3827,  -3690,  -3591,  -3464,  -3370,  -3250,  -3133,  -3047,
     -2935,  -2851,  -2742,  -2635,  -2566,  -2478,  -2389,  -2323,
     -2234,  -2168,  -2080,  -1991,  -1925,  -1836,  -1770,  -1681,
     -1593,  -1526,  -1438,  -1372,  -1283,  -1195,  -1138,  -1063,
     -1007,   -932,   -858,   -802,   -727,   -671,   -597,   -522,
      -466,   -391,   -335,   -261,   -186,   -130,    -55,      0
  },
  { // quantization_level=144
    -32767, -13114, -10892,  -9926,  -9061,  -8568,  -8006,  -7619,
     -7125,  -6839,  -6479,  -6166,  -5954,  -5693,  -5512,  -5284,
     -5121,  -4913,  -4769,  -4581,  -4402,  -4278,  -4120,  -4006,
     -3862,  -3757,  -3623,  -3528,  -3402,  -3281,  -3191,  -3076,
     -2990,  -2879,  -2796,  -2688,  -2611,  -2522,  -2434,  -2367,
     -2279,  -2212,  -2124,  -2057,  -1969,  -1903,  -1814,  -1726,
     -1659,  -1571,  -1504,  -1416,  -1349,  -1261,  -1194,  -1119,
     -1044,   -988,   -914,   -858,   -783,   -727,   -652,   -597,
      -522,   -447,   -391,   -317,   -261,   -186,   -130,    -55,
    0
  },
  { // quantization_level=146
    -32767, -13114, -10892,  -9926,  -9061,  -8568,  -8006,  -7619,
     -7125,  -6763,  -6479,  -6242,  -5954,  -5756,  -5512,  -5340,
     -5121,  -4965,  -4762,  -4627,  -4446,  -4319,  -4157,  -4043,
     -3897,  -3792,  -3657,  -3559,  -3433,  -3341,  -3220,  -3133,
     -3018,  -2935,  -2824,  -2742,  -2635,  -2566,  -2478,  -2411,
     -2323,  -2257,  -2168,  -2102,  -2013,  -1947,  -1858,  -1792,
     -1703,  -1637,  -1549,  -1482,  -1394,  -1327,  -1239,  -1175,
     -1100,  -1044,   -970,   -914,   -839,   -783,   -708,   -652,
      -578,   -522,   -447,   -391,   -317,   -261,   -186,   -130,
       -55,      0
  },
  { // quantization_level=148
    -32767, -13114, -11324,  -9926,  -9249,  -8568,  -8139,  -7619,
     -7248,  -6763,  -6566,  -6242,  -6022,  -5820,  -5571,  -5397,
     -5175,  -5017,  -4812,  -4671,  -4490,  -4361,  -4196,  -4082,
     -3932,  -3827,  -3723,  -3591,  -3495,  -3370,  -3281,  -3163,
     -3076,  -2963,  -2879,  -2769,  -2688,  -2611,  -2522,  -2456,
     -2367,  -2301,  -2212,  -2146,  -2057,  -1991,  -1903,  -1836,
     -1748,  -1681,  -1615,  -1526,  -1460,  -1372,  -1305,  -1217,
     -1156,  -1082,  -1026,   -951,   -895,   -820,   -764,   -708,
      -634,   -578,   -503,   -447,   -373,   -317,   -242,   -186,
      -111,    -55,      0
  },
  { // quantization_level=150
    -32767, -13114, -11324,  -9926,  -9249,  -8568,  -8139,  -7746,
```

Syntax

```
    -7248,  -6883,  -6566,  -6317,  -6094,  -5820,  -5632,  -5397,
    -5229,  -5017,  -4862,  -4719,  -4534,  -4402,  -4237,  -4120,
    -4006,  -3862,  -3757,  -3623,  -3528,  -3433,  -3311,  -3220,
    -3104,  -3018,  -2906,  -2824,  -2742,  -2635,  -2566,  -2478,
    -2411,  -2345,  -2257,  -2190,  -2102,  -2035,  -1947,  -1880,
    -1814,  -1726,  -1659,  -1571,  -1504,  -1438,  -1349,  -1283,
    -1195,  -1138,  -1082,  -1007,   -951,   -876,   -820,   -746,
     -690,   -634,   -559,   -503,   -429,   -373,   -317,   -242,
     -186,   -111,    -55,      0
  },
  { // quantization_level=152
    -32767, -13114, -11324,  -9926,  -9249,  -8723,  -8139,  -7746,
    -7370,  -6883,  -6652,  -6317,  -6094,  -5885,  -5632,  -5453,
    -5284,  -5069,  -4913,  -4769,  -4581,  -4446,  -4278,  -4157,
    -4043,  -3897,  -3792,  -3690,  -3559,  -3464,  -3341,  -3250,
    -3163,  -3047,  -2963,  -2879,  -2769,  -2688,  -2611,  -2522,
    -2456,  -2367,  -2301,  -2234,  -2146,  -2080,  -2013,  -1925,
    -1858,  -1770,  -1703,  -1637,  -1549,  -1482,  -1416,  -1327,
    -1261,  -1194,  -1119,  -1063,   -988,   -932,   -876,   -802,
     -746,   -690,   -615,   -559,   -485,   -429,   -373,   -298,
     -242,   -186,   -111,    -55,      0
  },
  { // quantization_level=154
    -32767, -13114, -11324, -10206,  -9249,  -8723,  -8278,  -7746,
    -7370,  -7004,  -6652,  -6398,  -6166,  -5885,  -5693,  -5512,
    -5284,  -5121,  -4965,  -4762,  -4627,  -4490,  -4319,  -4196,
    -4082,  -3932,  -3827,  -3723,  -3591,  -3495,  -3402,  -3281,
    -3191,  -3104,  -2990,  -2906,  -2824,  -2715,  -2635,  -2566,
    -2500,  -2411,  -2345,  -2279,  -2190,  -2124,  -2057,  -1969,
    -1903,  -1836,  -1748,  -1681,  -1615,  -1526,  -1460,  -1394,
    -1305,  -1239,  -1175,  -1100,  -1044,   -988,   -914,   -858,
     -802,   -727,   -671,   -615,   -541,   -485,   -429,   -354,
     -298,   -242,   -167,   -111,    -55,      0
  },
  { // quantization_level=156
    -32767, -13114, -11324, -10206,  -9249,  -8723,  -8278,  -7874,
    -7370,  -7004,  -6745,  -6398,  -6166,  -5954,  -5756,  -5512,
    -5340,  -5175,  -4965,  -4812,  -4671,  -4534,  -4361,  -4237,
    -4120,  -3969,  -3862,  -3757,  -3657,  -3528,  -3433,  -3341,
    -3220,  -3133,  -3047,  -2963,  -2851,  -2769,  -2688,  -2611,
    -2522,  -2456,  -2389,  -2301,  -2234,  -2168,  -2102,  -2013,
    -1947,  -1880,  -1792,  -1726,  -1659,  -1593,  -1504,  -1438,
    -1372,  -1283,  -1217,  -1156,  -1100,  -1026,   -970,   -914,
     -839,   -783,   -727,   -671,   -597,   -541,   -485,   -410,
     -354,   -298,   -242,   -167,   -111,    -55,      0
  },
  { // quantization_level=158
    -32767, -13114, -11324, -10206,  -9452,  -8723,  -8278,  -7874,
    -7494,  -7004,  -6745,  -6479,  -6242,  -5954,  -5756,  -5571,
    -5397,  -5175,  -5017,  -4862,  -4719,  -4534,  -4402,  -4278,
    -4157,  -4006,  -3897,  -3792,  -3690,  -3591,  -3464,  -3370,
    -3281,  -3191,  -3076,  -2990,  -2906,  -2824,  -2715,  -2635,
    -2566,  -2500,  -2411,  -2345,  -2279,  -2212,  -2124,  -2057,
    -1991,  -1925,  -1836,  -1770,  -1703,  -1637,  -1571,  -1482,
    -1416,  -1349,  -1283,  -1195,  -1138,  -1082,  -1026,   -951,
     -895,   -839,   -783,   -708,   -652,   -597,   -541,   -466,
     -410,   -354,   -298,   -223,   -167,   -111,    -55,      0
  },
  { // quantization_level=160
    -32767, -13114, -11324, -10206,  -9452,  -8886,  -8278,  -7874,
    -7494,  -7125,  -6839,  -6479,  -6242,  -6022,  -5820,  -5632,
    -5397,  -5229,  -5069,  -4913,  -4769,  -4581,  -4446,  -4319,
    -4196,  -4082,  -3932,  -3827,  -3723,  -3623,  -3528,  -3402,
    -3311,  -3220,  -3133,  -3047,  -2935,  -2851,  -2769,  -2688,
    -2611,  -2522,  -2456,  -2389,  -2323,  -2257,  -2168,  -2102,
    -2035,  -1969,  -1903,  -1814,  -1748,  -1681,  -1615,  -1549,
    -1460,  -1394,  -1327,  -1261,  -1194,  -1119,  -1063,  -1007,
     -951,   -895,   -820,   -764,   -708,   -652,   -597,   -522,
     -466,   -410,   -354,   -298,   -223,   -167,   -111,    -55,
    0
  },
  { // quantization_level=162
    -32767, -13114, -11324, -10206,  -9452,  -8886,  -8419,  -7874,
    -7494,  -7125,  -6763,  -6566,  -6317,  -6022,  -5820,  -5632,
    -5453,  -5284,  -5121,  -4913,  -4762,  -4627,  -4490,  -4361,
    -4237,  -4082,  -3969,  -3862,  -3757,  -3657,  -3559,  -3464,
    -3341,  -3250,  -3163,  -3076,  -2990,  -2906,  -2796,  -2715,
```

Syntax

```
    -2635,  -2566,  -2500,  -2434,  -2345,  -2279,  -2212,  -2146,
    -2080,  -2013,  -1925,  -1858,  -1792,  -1726,  -1659,  -1593,
    -1526,  -1438,  -1372,  -1305,  -1239,  -1175,  -1119,  -1044,
     -988,   -932,   -876,   -820,   -764,   -690,   -634,   -578,
     -522,   -466,   -410,   -335,   -279,   -223,   -167,   -111,
      -55,      0
  },
  { // quantization_level=164
    -32767, -13114, -11324, -10206,  -9452,  -8886,  -8419,  -8006,
     -7619,  -7125,  -6763,  -6566,  -6317,  -6094,  -5885,  -5693,
     -5512,  -5284,  -5121,  -4965,  -4812,  -4671,  -4534,  -4402,
     -4278,  -4120,  -4006,  -3897,  -3792,  -3690,  -3591,  -3495,
     -3402,  -3281,  -3191,  -3104,  -3018,  -2935,  -2851,  -2769,
     -2688,  -2611,  -2522,  -2456,  -2389,  -2323,  -2257,  -2190,
     -2124,  -2057,  -1969,  -1903,  -1836,  -1770,  -1703,  -1637,
     -1571,  -1504,  -1416,  -1349,  -1283,  -1217,  -1156,  -1100,
     -1044,   -988,   -914,   -858,   -802,   -746,   -690,   -634,
      -578,   -522,   -447,   -391,   -335,   -279,   -223,   -167,
      -111,    -55,      0
  },
  { // quantization_level=166
    -32767, -13114, -11324, -10206,  -9452,  -8886,  -8419,  -8006,
     -7619,  -7248,  -6883,  -6652,  -6317,  -6094,  -5885,  -5693,
     -5512,  -5340,  -5175,  -5017,  -4862,  -4719,  -4581,  -4446,
     -4278,  -4157,  -4043,  -3932,  -3827,  -3723,  -3623,  -3528,
     -3433,  -3341,  -3250,  -3163,  -3047,  -2963,  -2879,  -2796,
     -2715,  -2635,  -2566,  -2500,  -2434,  -2367,  -2301,  -2234,
     -2146,  -2080,  -2013,  -1947,  -1880,  -1814,  -1748,  -1681,
     -1615,  -1549,  -1482,  -1416,  -1327,  -1261,  -1195,  -1138,
     -1082,  -1026,   -970,   -914,   -858,   -802,   -746,   -690,
      -615,   -559,   -503,   -447,   -391,   -335,   -279,   -223,
      -167,   -111,    -55,      0
  },
  { // quantization_level=168
    -32767, -13114, -11324, -10206,  -9452,  -8886,  -8419,  -8006,
     -7619,  -7248,  -6883,  -6652,  -6398,  -6166,  -5954,  -5756,
     -5571,  -5397,  -5229,  -5069,  -4913,  -4769,  -4581,  -4446,
     -4319,  -4196,  -4082,  -3969,  -3862,  -3757,  -3657,  -3559,
     -3464,  -3370,  -3281,  -3191,  -3104,  -3018,  -2935,  -2851,
     -2769,  -2688,  -2611,  -2522,  -2456,  -2389,  -2323,  -2257,
     -2190,  -2124,  -2057,  -1991,  -1925,  -1858,  -1792,  -1726,
     -1659,  -1593,  -1526,  -1460,  -1394,  -1327,  -1261,  -1194,
     -1119,  -1063,  -1007,   -951,   -895,   -839,   -783,   -727,
      -671,   -615,   -559,   -503,   -447,   -391,   -335,   -279,
      -223,   -167,   -111,    -55,      0
  },
  { // quantization_level=170
    -32767, -13114, -11324, -10206,  -9452,  -8886,  -8419,  -8006,
     -7619,  -7248,  -6883,  -6652,  -6398,  -6166,  -5954,  -5756,
     -5571,  -5397,  -5229,  -5069,  -4913,  -4762,  -4627,  -4490,
     -4361,  -4237,  -4120,  -4006,  -3897,  -3792,  -3690,  -3591,
     -3495,  -3402,  -3311,  -3220,  -3133,  -3047,  -2963,  -2879,
     -2796,  -2715,  -2635,  -2566,  -2500,  -2434,  -2367,  -2301,
     -2234,  -2168,  -2102,  -2035,  -1969,  -1903,  -1836,  -1770,
     -1703,  -1637,  -1571,  -1504,  -1438,  -1372,  -1305,  -1239,
     -1175,  -1119,  -1063,  -1007,   -951,   -895,   -839,   -783,
      -727,   -671,   -615,   -559,   -503,   -447,   -391,   -335,
      -279,   -223,   -167,   -111,    -55,      0
  },
  { // quantization_level=172
    -32767, -13948, -11819, -10527,  -9678,  -9061,  -8568,  -8139,
     -7746,  -7370,  -7004,  -6745,  -6479,  -6242,  -6022,  -5820,
     -5632,  -5453,  -5284,  -5121,  -4965,  -4812,  -4671,  -4534,
     -4402,  -4278,  -4157,  -4043,  -3932,  -3827,  -3723,  -3623,
     -3528,  -3433,  -3341,  -3250,  -3163,  -3076,  -2990,  -2906,
     -2824,  -2742,  -2661,  -2611,  -2544,  -2478,  -2411,  -2345,
     -2279,  -2212,  -2146,  -2080,  -2013,  -1947,  -1880,  -1814,
     -1748,  -1681,  -1615,  -1549,  -1482,  -1416,  -1349,  -1283,
     -1217,  -1156,  -1100,  -1044,   -988,   -932,   -876,   -820,
      -764,   -708,   -652,   -597,   -541,   -485,   -429,   -373,
      -317,   -261,   -205,   -149,    -93,    -37,      0
  },
  { // quantization_level=174
    -32767, -13948, -11819, -10527,  -9678,  -9061,  -8568,  -8139,
     -7746,  -7370,  -7004,  -6745,  -6479,  -6242,  -6022,  -5820,
     -5632,  -5453,  -5340,  -5175,  -5017,  -4862,  -4719,  -4581,
     -4446,  -4319,  -4196,  -4082,  -3969,  -3862,  -3757,  -3657,
```

Syntax

```
    -3559,  -3464,  -3370,  -3311,  -3220,  -3133,  -3047,  -2963,
    -2879,  -2796,  -2715,  -2635,  -2566,  -2500,  -2434,  -2367,
    -2301,  -2234,  -2168,  -2102,  -2035,  -1991,  -1925,  -1858,
    -1792,  -1726,  -1659,  -1593,  -1526,  -1460,  -1394,  -1327,
    -1261,  -1195,  -1138,  -1082,  -1026,   -970,   -932,   -876,
     -820,   -764,   -708,   -652,   -597,   -541,   -485,   -429,
     -373,   -317,   -261,   -205,   -149,    -93,    -37,      0
  },
  { // quantization_level=176
    -32767, -13948, -11819, -10527,  -9678,  -9061,  -8568,  -8139,
    -7746,  -7370,  -7004,  -6839,  -6566,  -6317,  -6094,  -5885,
    -5693,  -5512,  -5340,  -5175,  -5017,  -4862,  -4769,  -4627,
    -4490,  -4361,  -4237,  -4120,  -4006,  -3897,  -3792,  -3690,
    -3591,  -3528,  -3433,  -3341,  -3250,  -3163,  -3076,  -2990,
    -2906,  -2824,  -2742,  -2661,  -2611,  -2544,  -2478,  -2411,
    -2345,  -2279,  -2212,  -2146,  -2080,  -2013,  -1947,  -1903,
    -1836,  -1770,  -1703,  -1637,  -1571,  -1504,  -1438,  -1372,
    -1305,  -1239,  -1194,  -1138,  -1082,  -1026,   -970,   -914,
     -858,   -802,   -746,   -690,   -634,   -597,   -541,   -485,
     -429,   -373,   -317,   -261,   -205,   -149,    -93,    -37,
      0
  },
  { // quantization_level=178
    -32767, -13948, -11819, -10527,  -9678,  -9061,  -8568,  -8139,
    -7746,  -7494,  -7125,  -6763,  -6566,  -6317,  -6094,  -5885,
    -5693,  -5571,  -5397,  -5229,  -5069,  -4913,  -4762,  -4627,
    -4490,  -4402,  -4278,  -4157,  -4043,  -3932,  -3827,  -3723,
    -3623,  -3559,  -3464,  -3370,  -3281,  -3191,  -3104,  -3018,
    -2935,  -2879,  -2796,  -2715,  -2635,  -2566,  -2500,  -2434,
    -2367,  -2323,  -2257,  -2190,  -2124,  -2057,  -1991,  -1925,
    -1858,  -1814,  -1748,  -1681,  -1615,  -1549,  -1482,  -1416,
    -1349,  -1305,  -1239,  -1175,  -1119,  -1063,  -1007,   -951,
     -895,   -858,   -802,   -746,   -690,   -634,   -578,   -522,
     -466,   -429,   -373,   -317,   -261,   -205,   -149,    -93,
      -37,      0
  },
  { // quantization_level=180
    -32767, -13948, -11819, -10527,  -9678,  -9061,  -8568,  -8278,
    -7874,  -7494,  -7125,  -6763,  -6566,  -6398,  -6166,  -5954,
    -5756,  -5571,  -5397,  -5229,  -5121,  -4965,  -4812,  -4671,
    -4534,  -4402,  -4319,  -4196,  -4082,  -3969,  -3862,  -3757,
    -3657,  -3591,  -3495,  -3402,  -3311,  -3220,  -3133,  -3076,
    -2990,  -2906,  -2824,  -2742,  -2661,  -2611,  -2544,  -2478,
    -2411,  -2345,  -2279,  -2212,  -2168,  -2102,  -2035,  -1969,
    -1903,  -1836,  -1792,  -1726,  -1659,  -1593,  -1526,  -1460,
    -1394,  -1349,  -1283,  -1217,  -1156,  -1100,  -1044,  -1007,
     -951,   -895,   -839,   -783,   -727,   -671,   -634,   -578,
     -522,   -466,   -410,   -354,   -317,   -261,   -205,   -149,
      -93,    -37,      0
  },
  { // quantization_level=182
    -32767, -13948, -11819, -10527,  -9678,  -9061,  -8723,  -8278,
    -7874,  -7494,  -7125,  -6883,  -6652,  -6398,  -6166,  -5954,
    -5756,  -5632,  -5453,  -5284,  -5121,  -4965,  -4862,  -4719,
    -4581,  -4446,  -4319,  -4237,  -4120,  -4006,  -3897,  -3792,
    -3690,  -3623,  -3528,  -3433,  -3341,  -3250,  -3191,  -3104,
    -3018,  -2935,  -2851,  -2796,  -2715,  -2635,  -2566,  -2500,
    -2434,  -2389,  -2323,  -2257,  -2190,  -2124,  -2080,  -2013,
    -1947,  -1880,  -1814,  -1770,  -1703,  -1637,  -1571,  -1504,
    -1438,  -1394,  -1327,  -1261,  -1195,  -1138,  -1100,  -1044,
     -988,   -932,   -876,   -839,   -783,   -727,   -671,   -615,
     -559,   -522,   -466,   -410,   -354,   -298,   -261,   -205,
     -149,    -93,    -37,      0
  },
  { // quantization_level=184
    -32767, -13948, -11819, -10527,  -9678,  -9249,  -8723,  -8278,
    -7874,  -7494,  -7248,  -6883,  -6652,  -6398,  -6242,  -6022,
    -5820,  -5632,  -5453,  -5340,  -5175,  -5017,  -4862,  -4769,
    -4627,  -4490,  -4361,  -4237,  -4157,  -4043,  -3932,  -3827,
    -3723,  -3657,  -3559,  -3464,  -3370,  -3311,  -3220,  -3133,
    -3047,  -2963,  -2906,  -2824,  -2742,  -2661,  -2611,  -2544,
    -2478,  -2411,  -2345,  -2301,  -2234,  -2168,  -2102,  -2035,
    -1991,  -1925,  -1858,  -1792,  -1748,  -1681,  -1615,  -1549,
    -1482,  -1438,  -1372,  -1305,  -1239,  -1194,  -1138,  -1082,
    -1026,   -970,   -932,   -876,   -820,   -764,   -708,   -671,
     -615,   -559,   -503,   -466,   -410,   -354,   -298,   -242,
     -205,   -149,    -93,    -37,      0
```

Syntax

```
},
{ // quantization_level=186
  -32767, -13948, -11819, -10527,  -9678,  -9249,  -8723,  -8278,
   -7874,  -7619,  -7248,  -6883,  -6652,  -6479,  -6242,  -6022,
   -5820,  -5693,  -5512,  -5340,  -5175,  -5069,  -4913,  -4762,
   -4627,  -4534,  -4402,  -4278,  -4157,  -4082,  -3969,  -3862,
   -3757,  -3690,  -3591,  -3495,  -3402,  -3341,  -3250,  -3163,
   -3076,  -3018,  -2935,  -2851,  -2769,  -2715,  -2635,  -2566,
   -2500,  -2456,  -2389,  -2323,  -2257,  -2212,  -2146,  -2080,
   -2013,  -1969,  -1903,  -1836,  -1770,  -1726,  -1659,  -1593,
   -1526,  -1482,  -1416,  -1349,  -1283,  -1239,  -1175,  -1119,
   -1063,  -1026,   -970,   -914,   -858,   -820,   -764,   -708,
    -652,   -615,   -559,   -503,   -447,   -410,   -354,   -298,
    -242,   -205,   -149,    -93,    -37,      0
},
{ // quantization_level=188
  -32767, -13948, -11819, -10527,  -9926,  -9249,  -8723,  -8278,
   -8006,  -7619,  -7248,  -7004,  -6745,  -6479,  -6242,  -6094,
   -5885,  -5693,  -5512,  -5397,  -5229,  -5069,  -4965,  -4812,
   -4671,  -4534,  -4446,  -4319,  -4196,  -4120,  -4006,  -3897,
   -3792,  -3723,  -3623,  -3528,  -3433,  -3370,  -3281,  -3191,
   -3133,  -3047,  -2963,  -2879,  -2824,  -2742,  -2661,  -2611,
   -2544,  -2478,  -2411,  -2367,  -2301,  -2234,  -2168,  -2124,
   -2057,  -1991,  -1947,  -1880,  -1814,  -1748,  -1703,  -1637,
   -1571,  -1504,  -1460,  -1394,  -1327,  -1283,  -1217,  -1156,
   -1100,  -1063,  -1007,   -951,   -914,   -858,   -802,   -746,
    -708,   -652,   -597,   -541,   -503,   -447,   -391,   -354,
    -298,   -242,   -186,   -149,    -93,    -37,      0
},
{ // quantization_level=190
  -32767, -13948, -11819, -10527,  -9926,  -9249,  -8723,  -8419,
   -8006,  -7619,  -7370,  -7004,  -6745,  -6479,  -6317,  -6094,
   -5885,  -5756,  -5571,  -5397,  -5284,  -5121,  -4965,  -4862,
   -4719,  -4581,  -4446,  -4361,  -4237,  -4120,  -4043,  -3932,
   -3827,  -3757,  -3657,  -3559,  -3464,  -3402,  -3311,  -3220,
   -3163,  -3076,  -2990,  -2935,  -2851,  -2769,  -2715,  -2635,
   -2566,  -2500,  -2456,  -2389,  -2323,  -2279,  -2212,  -2146,
   -2102,  -2035,  -1969,  -1925,  -1858,  -1792,  -1726,  -1681,
   -1615,  -1549,  -1504,  -1438,  -1372,  -1327,  -1261,  -1195,
   -1138,  -1100,  -1044,   -988,   -951,   -895,   -839,   -802,
    -746,   -690,   -652,   -597,   -541,   -485,   -447,   -391,
    -335,   -298,   -242,   -186,   -149,    -93,    -37,      0
},
{ // quantization_level=192
  -32767, -13948, -11819, -10892,  -9926,  -9249,  -8886,  -8419,
   -8006,  -7746,  -7370,  -7004,  -6839,  -6566,  -6317,  -6166,
   -5954,  -5756,  -5632,  -5453,  -5284,  -5175,  -5017,  -4862,
   -4769,  -4627,  -4490,  -4402,  -4278,  -4157,  -4082,  -3969,
   -3862,  -3792,  -3690,  -3591,  -3528,  -3433,  -3341,  -3281,
   -3191,  -3104,  -3047,  -2963,  -2879,  -2824,  -2742,  -2661,
   -2611,  -2544,  -2478,  -2434,  -2367,  -2301,  -2257,  -2190,
   -2124,  -2080,  -2013,  -1947,  -1903,  -1836,  -1770,  -1726,
   -1659,  -1593,  -1549,  -1482,  -1416,  -1372,  -1305,  -1239,
   -1194,  -1138,  -1082,  -1044,   -988,   -932,   -895,   -839,
    -783,   -746,   -690,   -634,   -597,   -541,   -485,   -447,
    -391,   -335,   -298,   -242,   -186,   -149,    -93,    -37,
    0
},
{ // quantization_level=194
  -32767, -13948, -11819, -10892,  -9926,  -9249,  -8886,  -8419,
   -8006,  -7746,  -7370,  -7004,  -6763,  -6566,  -6398,  -6166,
   -5954,  -5820,  -5632,  -5453,  -5340,  -5175,  -5017,  -4913,
   -4762,  -4671,  -4534,  -4402,  -4319,  -4196,  -4082,  -4006,
   -3897,  -3792,  -3723,  -3623,  -3528,  -3464,  -3370,  -3311,
   -3220,  -3133,  -3076,  -2990,  -2906,  -2851,  -2769,  -2688,
   -2635,  -2566,  -2522,  -2456,  -2389,  -2345,  -2279,  -2212,
   -2168,  -2102,  -2035,  -1991,  -1925,  -1880,  -1814,  -1748,
   -1703,  -1637,  -1571,  -1526,  -1460,  -1394,  -1349,  -1283,
   -1217,  -1175,  -1119,  -1082,  -1026,   -970,   -932,   -876,
    -820,   -783,   -727,   -671,   -634,   -578,   -541,   -485,
    -429,   -391,   -335,   -279,   -242,   -186,   -130,    -93,
     -37,      0
},
{ // quantization_level=196
  -32767, -13948, -11819, -10892,  -9926,  -9249,  -8886,  -8419,
   -8139,  -7746,  -7370,  -7125,  -6763,  -6652,  -6398,  -6166,
   -6022,  -5820,  -5632,  -5512,  -5340,  -5229,  -5069,  -4913,
```

Syntax

```
    -4812,  -4671,  -4581,  -4446,  -4319,  -4237,  -4120,  -4043,
    -3932,  -3827,  -3757,  -3657,  -3559,  -3495,  -3402,  -3341,
    -3250,  -3163,  -3104,  -3018,  -2963,  -2879,  -2796,  -2742,
    -2661,  -2611,  -2544,  -2478,  -2434,  -2367,  -2301,  -2257,
    -2190,  -2146,  -2080,  -2013,  -1969,  -1903,  -1858,  -1792,
    -1726,  -1681,  -1615,  -1549,  -1504,  -1438,  -1394,  -1327,
    -1261,  -1217,  -1156,  -1119,  -1063,  -1007,   -970,   -914,
     -876,   -820,   -764,   -727,   -671,   -615,   -578,   -522,
     -485,   -429,   -373,   -335,   -279,   -242,   -186,   -130,
      -93,    -37,      0
  },
  { // quantization_level=198
    -32767, -13948, -11819, -10892,  -9926,  -9452,  -8886,  -8419,
     -8139,  -7746,  -7494,  -7125,  -6763,  -6652,  -6398,  -6242,
     -6022,  -5885,  -5693,  -5512,  -5397,  -5229,  -5121,  -4965,
     -4812,  -4719,  -4581,  -4490,  -4361,  -4278,  -4157,  -4043,
     -3969,  -3862,  -3792,  -3690,  -3591,  -3528,  -3433,  -3370,
     -3281,  -3191,  -3133,  -3047,  -2990,  -2906,  -2851,  -2769,
     -2688,  -2635,  -2566,  -2522,  -2456,  -2389,  -2345,  -2279,
     -2234,  -2168,  -2124,  -2057,  -1991,  -1947,  -1880,  -1836,
     -1770,  -1703,  -1659,  -1593,  -1549,  -1482,  -1416,  -1372,
     -1305,  -1261,  -1195,  -1156,  -1100,  -1044,  -1007,   -951,
      -914,   -858,   -802,   -764,   -708,   -671,   -615,   -578,
      -522,   -466,   -429,   -373,   -335,   -279,   -223,   -186,
      -130,    -93,    -37,      0
  },
  { // quantization_level=200
    -32767, -13948, -11819, -10892,  -9926,  -9452,  -8886,  -8568,
     -8139,  -7746,  -7494,  -7125,  -6883,  -6652,  -6479,  -6242,
     -6094,  -5885,  -5693,  -5571,  -5397,  -5284,  -5121,  -5017,
     -4862,  -4769,  -4627,  -4490,  -4402,  -4278,  -4196,  -4082,
     -4006,  -3897,  -3792,  -3723,  -3623,  -3559,  -3464,  -3402,
     -3311,  -3250,  -3163,  -3076,  -3018,  -2935,  -2879,  -2796,
     -2742,  -2661,  -2611,  -2544,  -2478,  -2434,  -2367,  -2323,
     -2257,  -2212,  -2146,  -2080,  -2035,  -1969,  -1925,  -1858,
     -1814,  -1748,  -1703,  -1637,  -1571,  -1526,  -1460,  -1416,
     -1349,  -1305,  -1239,  -1194,  -1138,  -1082,  -1044,   -988,
      -951,   -895,   -858,   -802,   -746,   -708,   -652,   -615,
      -559,   -522,   -466,   -429,   -373,   -317,   -279,   -223,
      -186,   -130,    -93,    -37,      0
  },
  { // quantization_level=202
    -32767, -13948, -11819, -10892,  -9926,  -9452,  -8886,  -8568,
     -8139,  -7874,  -7494,  -7248,  -6883,  -6745,  -6479,  -6242,
     -6094,  -5885,  -5756,  -5571,  -5453,  -5284,  -5175,  -5017,
     -4913,  -4762,  -4671,  -4534,  -4446,  -4319,  -4196,  -4120,
     -4006,  -3932,  -3827,  -3757,  -3657,  -3591,  -3495,  -3433,
     -3341,  -3281,  -3191,  -3133,  -3047,  -2963,  -2906,  -2824,
     -2769,  -2688,  -2635,  -2566,  -2522,  -2456,  -2411,  -2345,
     -2301,  -2234,  -2168,  -2124,  -2057,  -2013,  -1947,  -1903,
     -1836,  -1792,  -1726,  -1681,  -1615,  -1571,  -1504,  -1460,
     -1394,  -1327,  -1283,  -1217,  -1175,  -1119,  -1082,  -1026,
      -988,   -932,   -895,   -839,   -802,   -746,   -708,   -652,
      -597,   -559,   -503,   -466,   -410,   -373,   -317,   -279,
      -223,   -186,   -130,    -93,    -37,      0
  },
  { // quantization_level=204
    -32767, -13948, -11819, -10892,  -9926,  -9452,  -8886,  -8568,
     -8139,  -7874,  -7494,  -7248,  -6883,  -6745,  -6479,  -6317,
     -6094,  -5954,  -5756,  -5632,  -5453,  -5340,  -5175,  -5069,
     -4913,  -4812,  -4671,  -4581,  -4446,  -4361,  -4237,  -4157,
     -4043,  -3969,  -3862,  -3792,  -3690,  -3623,  -3528,  -3464,
     -3370,  -3311,  -3220,  -3163,  -3076,  -3018,  -2935,  -2879,
     -2796,  -2742,  -2661,  -2611,  -2544,  -2478,  -2434,  -2367,
     -2323,  -2257,  -2212,  -2146,  -2102,  -2035,  -1991,  -1925,
     -1880,  -1814,  -1770,  -1703,  -1659,  -1593,  -1549,  -1482,
     -1438,  -1372,  -1327,  -1261,  -1217,  -1156,  -1119,  -1063,
     -1026,   -970,   -932,   -876,   -839,   -783,   -746,   -690,
      -652,   -597,   -559,   -503,   -466,   -410,   -373,   -317,
      -279,   -223,   -186,   -130,    -93,    -37,      0
  },
  { // quantization_level=206
    -32767, -13948, -12403, -10892, -10206,  -9452,  -9061,  -8568,
     -8278,  -7874,  -7619,  -7248,  -7004,  -6745,  -6566,  -6317,
     -6166,  -5954,  -5820,  -5632,  -5512,  -5340,  -5229,  -5069,
     -4965,  -4812,  -4719,  -4581,  -4490,  -4361,  -4278,  -4157,
     -4082,  -3969,  -3897,  -3827,  -3723,  -3657,  -3559,  -3495,
```

Syntax

```
    -3402,  -3341,  -3250,  -3191,  -3104,  -3047,  -2963,  -2906,
    -2824,  -2769,  -2688,  -2635,  -2566,  -2522,  -2456,  -2411,
    -2345,  -2301,  -2234,  -2190,  -2124,  -2080,  -2013,  -1969,
    -1903,  -1858,  -1792,  -1748,  -1681,  -1637,  -1593,  -1526,
    -1482,  -1416,  -1372,  -1305,  -1261,  -1195,  -1156,  -1100,
    -1063,  -1007,   -970,   -914,   -876,   -820,   -783,   -727,
     -690,   -634,   -597,   -541,   -503,   -447,   -410,   -354,
     -317,   -261,   -223,   -167,   -130,    -74,    -37,      0
  },
  { // quantization_level=208
    -32767, -13948, -12403, -10892, -10206,  -9452,  -9061,  -8568,
    -8278,  -7874,  -7619,  -7248,  -7004,  -6839,  -6566,  -6398,
    -6166,  -6022,  -5820,  -5693,  -5512,  -5397,  -5229,  -5121,
    -4965,  -4862,  -4769,  -4627,  -4534,  -4402,  -4319,  -4196,
    -4120,  -4006,  -3932,  -3827,  -3757,  -3657,  -3591,  -3528,
    -3433,  -3370,  -3281,  -3220,  -3133,  -3076,  -2990,  -2935,
    -2851,  -2796,  -2715,  -2661,  -2611,  -2544,  -2500,  -2434,
    -2389,  -2323,  -2279,  -2212,  -2168,  -2102,  -2057,  -1991,
    -1947,  -1903,  -1836,  -1792,  -1726,  -1681,  -1615,  -1571,
    -1504,  -1460,  -1394,  -1349,  -1283,  -1239,  -1194,  -1138,
    -1100,  -1044,  -1007,   -951,   -914,   -858,   -820,   -764,
     -727,   -671,   -634,   -597,   -541,   -503,   -447,   -410,
     -354,   -317,   -261,   -223,   -167,   -130,    -74,    -37,
      0
  },
  { // quantization_level=210
    -32767, -13948, -12403, -10892, -10206,  -9452,  -9061,  -8568,
    -8278,  -8006,  -7619,  -7370,  -7004,  -6763,  -6566,  -6398,
    -6166,  -6022,  -5885,  -5693,  -5571,  -5397,  -5284,  -5121,
    -5017,  -4913,  -4762,  -4671,  -4534,  -4446,  -4319,  -4237,
    -4120,  -4043,  -3969,  -3862,  -3792,  -3690,  -3623,  -3528,
    -3464,  -3402,  -3311,  -3250,  -3163,  -3104,  -3018,  -2963,
    -2879,  -2824,  -2769,  -2688,  -2635,  -2566,  -2522,  -2456,
    -2411,  -2367,  -2301,  -2257,  -2190,  -2146,  -2080,  -2035,
    -1969,  -1925,  -1880,  -1814,  -1770,  -1703,  -1659,  -1593,
    -1549,  -1504,  -1438,  -1394,  -1327,  -1283,  -1217,  -1175,
    -1119,  -1082,  -1044,   -988,   -951,   -895,   -858,   -802,
     -764,   -727,   -671,   -634,   -578,   -541,   -485,   -447,
     -391,   -354,   -317,   -261,   -223,   -167,   -130,    -74,
      -37,      0
  },
  { // quantization_level=212
    -32767, -13948, -12403, -10892, -10206,  -9452,  -9061,  -8723,
    -8278,  -8006,  -7619,  -7370,  -7125,  -6763,  -6652,  -6398,
    -6242,  -6022,  -5885,  -5756,  -5571,  -5453,  -5284,  -5175,
    -5069,  -4913,  -4812,  -4671,  -4581,  -4446,  -4361,  -4278,
    -4157,  -4082,  -3969,  -3897,  -3827,  -3723,  -3657,  -3559,
    -3495,  -3402,  -3341,  -3281,  -3191,  -3133,  -3047,  -2990,
    -2935,  -2851,  -2796,  -2715,  -2661,  -2611,  -2544,  -2500,
    -2434,  -2389,  -2323,  -2279,  -2234,  -2168,  -2124,  -2057,
    -2013,  -1969,  -1903,  -1858,  -1792,  -1748,  -1681,  -1637,
    -1593,  -1526,  -1482,  -1416,  -1372,  -1327,  -1261,  -1217,
    -1156,  -1119,  -1063,  -1026,   -988,   -932,   -895,   -839,
     -802,   -764,   -708,   -671,   -615,   -578,   -522,   -485,
     -447,   -391,   -354,   -298,   -261,   -223,   -167,   -130,
      -74,    -37,      0
  },
  { // quantization_level=214
    -32767, -13948, -12403, -10892, -10206,  -9678,  -9061,  -8723,
    -8278,  -8006,  -7746,  -7370,  -7125,  -6763,  -6652,  -6479,
    -6242,  -6094,  -5885,  -5756,  -5632,  -5453,  -5340,  -5175,
    -5069,  -4965,  -4812,  -4719,  -4627,  -4490,  -4402,  -4278,
    -4196,  -4120,  -4006,  -3932,  -3827,  -3757,  -3690,  -3591,
    -3528,  -3433,  -3370,  -3311,  -3220,  -3163,  -3076,  -3018,
    -2963,  -2879,  -2824,  -2742,  -2688,  -2635,  -2566,  -2522,
    -2478,  -2411,  -2367,  -2301,  -2257,  -2212,  -2146,  -2102,
    -2035,  -1991,  -1947,  -1880,  -1836,  -1770,  -1726,  -1681,
    -1615,  -1571,  -1504,  -1460,  -1416,  -1349,  -1305,  -1239,
    -1195,  -1156,  -1100,  -1063,  -1026,   -970,   -932,   -876,
     -839,   -802,   -746,   -708,   -652,   -615,   -578,   -522,
     -485,   -429,   -391,   -354,   -298,   -261,   -205,   -167,
     -130,    -74,    -37,      0
  },
  { // quantization_level=216
    -32767, -13948, -12403, -10892, -10206,  -9678,  -9061,  -8723,
    -8419,  -8006,  -7746,  -7370,  -7125,  -6883,  -6652,  -6479,
    -6317,  -6094,  -5954,  -5756,  -5632,  -5512,  -5340,  -5229,
```

Syntax

```
    -5121,  -4965,  -4862,  -4769,  -4627,  -4534,  -4402,  -4319,
    -4237,  -4120,  -4043,  -3969,  -3862,  -3792,  -3690,  -3623,
    -3559,  -3464,  -3402,  -3341,  -3250,  -3191,  -3104,  -3047,
    -2990,  -2906,  -2851,  -2796,  -2715,  -2661,  -2611,  -2544,
    -2500,  -2434,  -2389,  -2345,  -2279,  -2234,  -2190,  -2124,
    -2080,  -2013,  -1969,  -1925,  -1858,  -1814,  -1770,  -1703,
    -1659,  -1593,  -1549,  -1504,  -1438,  -1394,  -1349,  -1283,
    -1239,  -1194,  -1138,  -1100,  -1044,  -1007,   -970,   -914,
     -876,   -839,   -783,   -746,   -690,   -652,   -615,   -559,
     -522,   -485,   -429,   -391,   -335,   -298,   -261,   -205,
     -167,   -130,    -74,    -37,      0
  },
  { // quantization_level=218
    -32767, -13948, -12403, -10892, -10206,  -9678,  -9061,  -8723,
     -8419,  -8006,  -7746,  -7494,  -7125,  -6883,  -6745,  -6479,
     -6317,  -6166,  -5954,  -5820,  -5693,  -5512,  -5397,  -5229,
     -5121,  -5017,  -4862,  -4762,  -4671,  -4534,  -4446,  -4361,
     -4237,  -4157,  -4082,  -3969,  -3897,  -3827,  -3723,  -3657,
     -3591,  -3495,  -3433,  -3370,  -3281,  -3220,  -3133,  -3076,
     -3018,  -2935,  -2879,  -2824,  -2742,  -2688,  -2635,  -2566,
     -2522,  -2478,  -2411,  -2367,  -2323,  -2257,  -2212,  -2168,
     -2102,  -2057,  -1991,  -1947,  -1903,  -1836,  -1792,  -1748,
     -1681,  -1637,  -1593,  -1526,  -1482,  -1438,  -1372,  -1327,
     -1283,  -1217,  -1175,  -1138,  -1082,  -1044,  -1007,   -951,
      -914,   -858,   -820,   -783,   -727,   -690,   -652,   -597,
      -559,   -522,   -466,   -429,   -391,   -335,   -298,   -261,
      -205,   -167,   -130,    -74,    -37,      0
  },
  { // quantization_level=220
    -32767, -13948, -12403, -11324, -10206,  -9678,  -9249,  -8723,
     -8419,  -8139,  -7746,  -7494,  -7248,  -6883,  -6745,  -6566,
     -6317,  -6166,  -6022,  -5820,  -5693,  -5571,  -5397,  -5284,
     -5175,  -5017,  -4913,  -4812,  -4671,  -4581,  -4490,  -4361,
     -4278,  -4196,  -4082,  -4006,  -3932,  -3827,  -3757,  -3690,
     -3591,  -3528,  -3464,  -3370,  -3311,  -3250,  -3163,  -3104,
     -3047,  -2963,  -2906,  -2851,  -2769,  -2715,  -2661,  -2611,
     -2544,  -2500,  -2456,  -2389,  -2345,  -2301,  -2234,  -2190,
     -2146,  -2080,  -2035,  -1991,  -1925,  -1880,  -1836,  -1770,
     -1726,  -1681,  -1615,  -1571,  -1526,  -1460,  -1416,  -1372,
     -1305,  -1261,  -1217,  -1156,  -1119,  -1082,  -1026,   -988,
      -951,   -895,   -858,   -820,   -764,   -727,   -690,   -634,
      -597,   -559,   -503,   -466,   -429,   -373,   -335,   -298,
      -242,   -205,   -167,   -111,    -74,    -37,      0
  },
  { // quantization_level=222
    -32767, -13948, -12403, -11324, -10206,  -9678,  -9249,  -8723,
     -8419,  -8139,  -7746,  -7494,  -7248,  -7004,  -6745,  -6566,
     -6398,  -6166,  -6022,  -5885,  -5693,  -5571,  -5453,  -5284,
     -5175,  -5069,  -4965,  -4812,  -4719,  -4627,  -4490,  -4402,
     -4319,  -4196,  -4120,  -4043,  -3932,  -3862,  -3792,  -3723,
     -3623,  -3559,  -3495,  -3402,  -3341,  -3281,  -3191,  -3133,
     -3076,  -2990,  -2935,  -2879,  -2824,  -2742,  -2688,  -2635,
     -2566,  -2522,  -2478,  -2411,  -2367,  -2323,  -2279,  -2212,
     -2168,  -2124,  -2057,  -2013,  -1969,  -1903,  -1858,  -1814,
     -1748,  -1703,  -1659,  -1615,  -1549,  -1504,  -1460,  -1394,
     -1349,  -1305,  -1239,  -1195,  -1156,  -1100,  -1063,  -1026,
      -988,   -932,   -895,   -858,   -802,   -764,   -727,   -671,
      -634,   -597,   -541,   -503,   -466,   -429,   -373,   -335,
      -298,   -242,   -205,   -167,   -111,    -74,    -37,      0
  },
  { // quantization_level=224
    -32767, -13948, -12403, -11324, -10206,  -9678,  -9249,  -8886,
     -8419,  -8139,  -7874,  -7494,  -7248,  -7004,  -6839,  -6566,
     -6398,  -6242,  -6022,  -5885,  -5756,  -5632,  -5453,  -5340,
     -5229,  -5069,  -4965,  -4862,  -4769,  -4627,  -4534,  -4446,
     -4319,  -4237,  -4157,  -4082,  -3969,  -3897,  -3827,  -3723,
     -3657,  -3591,  -3528,  -3433,  -3370,  -3311,  -3220,  -3163,
     -3104,  -3047,  -2963,  -2906,  -2851,  -2769,  -2715,  -2661,
     -2611,  -2544,  -2500,  -2456,  -2389,  -2345,  -2301,  -2257,
     -2190,  -2146,  -2102,  -2035,  -1991,  -1947,  -1903,  -1836,
     -1792,  -1748,  -1681,  -1637,  -1593,  -1549,  -1482,  -1438,
     -1394,  -1327,  -1283,  -1239,  -1194,  -1138,  -1100,  -1063,
     -1007,   -970,   -932,   -895,   -839,   -802,   -764,   -708,
      -671,   -634,   -597,   -541,   -503,   -466,   -410,   -373,
      -335,   -298,   -242,   -205,   -167,   -111,    -74,    -37,
     0
  },
```

Syntax

```
{ // quantization_level=226
  -32767, -13948, -12403, -11324, -10206,  -9678,  -9249,  -8886,
   -8419,  -8139,  -7874,  -7619,  -7248,  -7004,  -6763,  -6652,
   -6398,  -6242,  -6094,  -5885,  -5756,  -5632,  -5512,  -5340,
   -5229,  -5121,  -5017,  -4862,  -4762,  -4671,  -4581,  -4446,
   -4361,  -4278,  -4157,  -4082,  -4006,  -3932,  -3827,  -3757,
   -3690,  -3623,  -3528,  -3464,  -3402,  -3341,  -3250,  -3191,
   -3133,  -3047,  -2990,  -2935,  -2879,  -2796,  -2742,  -2688,
   -2635,  -2566,  -2522,  -2478,  -2434,  -2367,  -2323,  -2279,
   -2234,  -2168,  -2124,  -2080,  -2013,  -1969,  -1925,  -1880,
   -1814,  -1770,  -1726,  -1681,  -1615,  -1571,  -1526,  -1482,
   -1416,  -1372,  -1327,  -1261,  -1217,  -1175,  -1138,  -1082,
   -1044,  -1007,   -970,   -914,   -876,   -839,   -802,   -746,
    -708,   -671,   -615,   -578,   -541,   -503,   -447,   -410,
    -373,   -335,   -279,   -242,   -205,   -167,   -111,    -74,
     -37,      0
},
{ // quantization_level=228
  -32767, -13948, -12403, -11324, -10527,  -9678,  -9249,  -8886,
   -8568,  -8139,  -7874,  -7619,  -7370,  -7004,  -6763,  -6652,
   -6479,  -6242,  -6094,  -5954,  -5820,  -5632,  -5512,  -5397,
   -5284,  -5121,  -5017,  -4913,  -4812,  -4671,  -4581,  -4490,
   -4402,  -4278,  -4196,  -4120,  -4043,  -3932,  -3862,  -3792,
   -3723,  -3623,  -3559,  -3495,  -3433,  -3341,  -3281,  -3220,
   -3163,  -3076,  -3018,  -2963,  -2906,  -2824,  -2769,  -2715,
   -2661,  -2611,  -2544,  -2500,  -2456,  -2411,  -2345,  -2301,
   -2257,  -2212,  -2146,  -2102,  -2057,  -2013,  -1947,  -1903,
   -1858,  -1814,  -1748,  -1703,  -1659,  -1615,  -1549,  -1504,
   -1460,  -1416,  -1349,  -1305,  -1261,  -1217,  -1156,  -1119,
   -1082,  -1044,   -988,   -951,   -914,   -876,   -820,   -783,
    -746,   -708,   -652,   -615,   -578,   -541,   -485,   -447,
    -410,   -373,   -317,   -279,   -242,   -205,   -149,   -111,
     -74,    -37,      0
},
{ // quantization_level=230
  -32767, -13948, -12403, -11324, -10527,  -9678,  -9249,  -8886,
   -8568,  -8139,  -7874,  -7619,  -7370,  -7125,  -6763,  -6652,
   -6479,  -6317,  -6094,  -5954,  -5820,  -5693,  -5571,  -5397,
   -5284,  -5175,  -5069,  -4913,  -4812,  -4719,  -4627,  -4490,
   -4402,  -4319,  -4237,  -4157,  -4043,  -3969,  -3897,  -3827,
   -3723,  -3657,  -3591,  -3528,  -3464,  -3370,  -3311,  -3250,
   -3191,  -3104,  -3047,  -2990,  -2935,  -2879,  -2796,  -2742,
   -2688,  -2635,  -2566,  -2522,  -2478,  -2434,  -2367,  -2323,
   -2279,  -2234,  -2190,  -2124,  -2080,  -2035,  -1991,  -1925,
   -1880,  -1836,  -1792,  -1748,  -1681,  -1637,  -1593,  -1549,
   -1482,  -1438,  -1394,  -1349,  -1305,  -1239,  -1195,  -1156,
   -1119,  -1063,  -1026,   -988,   -951,   -895,   -858,   -820,
    -783,   -746,   -690,   -652,   -615,   -578,   -522,   -485,
    -447,   -410,   -373,   -317,   -279,   -242,   -205,   -149,
    -111,    -74,    -37,      0
},
{ // quantization_level=232
  -32767, -13948, -12403, -11324, -10527,  -9678,  -9249,  -8886,
   -8568,  -8278,  -7874,  -7619,  -7370,  -7125,  -6883,  -6652,
   -6479,  -6317,  -6166,  -6022,  -5820,  -5693,  -5571,  -5453,
   -5340,  -5175,  -5069,  -4965,  -4862,  -4769,  -4627,  -4534,
   -4446,  -4361,  -4237,  -4157,  -4082,  -4006,  -3932,  -3827,
   -3757,  -3690,  -3623,  -3559,  -3464,  -3402,  -3341,  -3281,
   -3220,  -3133,  -3076,  -3018,  -2963,  -2906,  -2824,  -2769,
   -2715,  -2661,  -2611,  -2544,  -2500,  -2456,  -2411,  -2345,
   -2301,  -2257,  -2212,  -2168,  -2102,  -2057,  -2013,  -1969,
   -1925,  -1858,  -1814,  -1770,  -1726,  -1681,  -1615,  -1571,
   -1526,  -1482,  -1438,  -1372,  -1327,  -1283,  -1239,  -1194,
   -1138,  -1100,  -1063,  -1026,   -970,   -932,   -895,   -858,
    -820,   -764,   -727,   -690,   -652,   -615,   -559,   -522,
    -485,   -447,   -410,   -354,   -317,   -279,   -242,   -205,
    -149,   -111,    -74,    -37,      0
},
{ // quantization_level=234
  -32767, -13948, -12403, -11324, -10527,  -9926,  -9249,  -8886,
   -8568,  -8278,  -8006,  -7619,  -7370,  -7125,  -6883,  -6745,
   -6479,  -6317,  -6166,  -6022,  -5885,  -5756,  -5571,  -5453,
   -5340,  -5229,  -5121,  -4965,  -4862,  -4762,  -4671,  -4581,
   -4446,  -4361,  -4278,  -4196,  -4120,  -4043,  -3932,  -3862,
   -3792,  -3723,  -3657,  -3559,  -3495,  -3433,  -3370,  -3311,
   -3220,  -3163,  -3104,  -3047,  -2990,  -2935,  -2851,  -2796,
   -2742,  -2688,  -2635,  -2566,  -2522,  -2478,  -2434,  -2389,
```

Syntax

```
    -2323,  -2279,  -2234,  -2190,  -2146,  -2102,  -2035,  -1991,
    -1947,  -1903,  -1858,  -1792,  -1748,  -1703,  -1659,  -1615,
    -1549,  -1504,  -1460,  -1416,  -1372,  -1327,  -1261,  -1217,
    -1175,  -1138,  -1100,  -1044,  -1007,   -970,   -932,   -895,
     -839,   -802,   -764,   -727,   -690,   -652,   -597,   -559,
     -522,   -485,   -447,   -391,   -354,   -317,   -279,   -242,
     -186,   -149,   -111,    -74,    -37,      0
  },
  { // quantization_level=236
    -32767, -13948, -12403, -11324, -10527,  -9926,  -9249,  -8886,
     -8568,  -8278,  -8006,  -7746,  -7370,  -7125,  -6883,  -6745,
     -6566,  -6398,  -6166,  -6022,  -5885,  -5756,  -5632,  -5512,
     -5340,  -5229,  -5121,  -5017,  -4913,  -4812,  -4671,  -4581,
     -4490,  -4402,  -4319,  -4237,  -4120,  -4043,  -3969,  -3897,
     -3827,  -3757,  -3657,  -3591,  -3528,  -3464,  -3402,  -3341,
     -3250,  -3191,  -3133,  -3076,  -3018,  -2963,  -2879,  -2824,
     -2769,  -2715,  -2661,  -2611,  -2544,  -2500,  -2456,  -2411,
     -2367,  -2301,  -2257,  -2212,  -2168,  -2124,  -2080,  -2013,
     -1969,  -1925,  -1880,  -1836,  -1792,  -1726,  -1681,  -1637,
     -1593,  -1549,  -1504,  -1438,  -1394,  -1349,  -1305,  -1261,
     -1217,  -1156,  -1119,  -1082,  -1044,  -1007,   -970,   -914,
      -876,   -839,   -802,   -764,   -727,   -671,   -634,   -597,
      -559,   -522,   -485,   -429,   -391,   -354,   -317,   -279,
      -242,   -186,   -149,   -111,    -74,    -37,      0
  },
  { // quantization_level=238
    -32767, -13948, -12403, -11324, -10527,  -9926,  -9452,  -8886,
     -8568,  -8278,  -8006,  -7746,  -7494,  -7248,  -6883,  -6745,
     -6566,  -6398,  -6242,  -6094,  -5885,  -5756,  -5632,  -5512,
     -5397,  -5284,  -5175,  -5017,  -4913,  -4812,  -4719,  -4627,
     -4534,  -4446,  -4319,  -4237,  -4157,  -4082,  -4006,  -3932,
     -3827,  -3757,  -3690,  -3623,  -3559,  -3495,  -3433,  -3341,
     -3281,  -3220,  -3163,  -3104,  -3047,  -2963,  -2906,  -2851,
     -2796,  -2742,  -2688,  -2635,  -2566,  -2522,  -2478,  -2434,
     -2389,  -2345,  -2301,  -2234,  -2190,  -2146,  -2102,  -2057,
     -2013,  -1947,  -1903,  -1858,  -1814,  -1770,  -1726,  -1681,
     -1615,  -1571,  -1526,  -1482,  -1438,  -1394,  -1327,  -1283,
     -1239,  -1195,  -1156,  -1119,  -1082,  -1026,   -988,   -951,
      -914,   -876,   -839,   -802,   -746,   -708,   -671,   -634,
      -597,   -559,   -503,   -466,   -429,   -391,   -354,   -317,
      -279,   -223,   -186,   -149,   -111,    -74,    -37,      0
  },
  { // quantization_level=240
    -32767, -13948, -12403, -11324, -10527,  -9926,  -9452,  -9061,
     -8568,  -8278,  -8006,  -7746,  -7494,  -7248,  -7004,  -6839,
     -6566,  -6398,  -6242,  -6094,  -5954,  -5820,  -5693,  -5512,
     -5397,  -5284,  -5175,  -5069,  -4965,  -4862,  -4769,  -4627,
     -4534,  -4446,  -4361,  -4278,  -4196,  -4120,  -4006,  -3932,
     -3862,  -3792,  -3723,  -3657,  -3591,  -3528,  -3433,  -3370,
     -3311,  -3250,  -3191,  -3133,  -3076,  -2990,  -2935,  -2879,
     -2824,  -2769,  -2715,  -2661,  -2611,  -2544,  -2500,  -2456,
     -2411,  -2367,  -2323,  -2279,  -2212,  -2168,  -2124,  -2080,
     -2035,  -1991,  -1947,  -1903,  -1836,  -1792,  -1748,  -1703,
     -1659,  -1615,  -1571,  -1504,  -1460,  -1416,  -1372,  -1327,
     -1283,  -1239,  -1194,  -1138,  -1100,  -1063,  -1026,   -988,
      -951,   -914,   -858,   -820,   -783,   -746,   -708,   -671,
      -634,   -597,   -541,   -503,   -466,   -429,   -391,   -354,
      -317,   -261,   -223,   -186,   -149,   -111,    -74,    -37,
     0
  },
  { // quantization_level=242
    -32767, -13948, -12403, -11324, -10527,  -9926,  -9452,  -9061,
     -8723,  -8278,  -8006,  -7746,  -7494,  -7248,  -7004,  -6763,
     -6652,  -6479,  -6242,  -6094,  -5954,  -5820,  -5693,  -5571,
     -5453,  -5340,  -5175,  -5069,  -4965,  -4862,  -4762,  -4671,
     -4581,  -4490,  -4402,  -4278,  -4196,  -4120,  -4043,  -3969,
     -3897,  -3827,  -3757,  -3690,  -3591,  -3528,  -3464,  -3402,
     -3341,  -3281,  -3220,  -3163,  -3076,  -3018,  -2963,  -2906,
     -2851,  -2796,  -2742,  -2688,  -2635,  -2566,  -2522,  -2478,
     -2434,  -2389,  -2345,  -2301,  -2257,  -2212,  -2146,  -2102,
     -2057,  -2013,  -1969,  -1925,  -1880,  -1836,  -1770,  -1726,
     -1681,  -1637,  -1593,  -1549,  -1504,  -1460,  -1416,  -1349,
     -1305,  -1261,  -1217,  -1175,  -1138,  -1100,  -1063,  -1026,
      -970,   -932,   -895,   -858,   -820,   -783,   -746,   -708,
      -652,   -615,   -578,   -541,   -503,   -466,   -429,   -391,
      -354,   -298,   -261,   -223,   -186,   -149,   -111,    -74,
       -37,      0
```

Syntax

```
},
{ // quantization_level=244
  -32767, -13948, -12403, -11324, -10527,  -9926,  -9452,  -9061,
   -8723,  -8419,  -8139,  -7746,  -7494,  -7248,  -7004,  -6763,
   -6652,  -6479,  -6317,  -6166,  -6022,  -5820,  -5693,  -5571,
   -5453,  -5340,  -5229,  -5121,  -5017,  -4913,  -4812,  -4671,
   -4581,  -4490,  -4402,  -4319,  -4237,  -4157,  -4082,  -4006,
   -3932,  -3827,  -3757,  -3690,  -3623,  -3559,  -3495,  -3433,
   -3370,  -3311,  -3250,  -3163,  -3104,  -3047,  -2990,  -2935,
   -2879,  -2824,  -2769,  -2715,  -2661,  -2611,  -2544,  -2500,
   -2456,  -2411,  -2367,  -2323,  -2279,  -2234,  -2190,  -2146,
   -2080,  -2035,  -1991,  -1947,  -1903,  -1858,  -1814,  -1770,
   -1726,  -1681,  -1615,  -1571,  -1526,  -1482,  -1438,  -1394,
   -1349,  -1305,  -1261,  -1217,  -1156,  -1119,  -1082,  -1044,
   -1007,   -970,   -932,   -895,   -858,   -820,   -764,   -727,
    -690,   -652,   -615,   -578,   -541,   -503,   -466,   -429,
    -373,   -335,   -298,   -261,   -223,   -186,   -149,   -111,
     -74,    -37,      0
},
{ // quantization_level=246
  -32767, -13948, -12403, -11324, -10527,  -9926,  -9452,  -9061,
   -8723,  -8419,  -8139,  -7874,  -7619,  -7248,  -7004,  -6763,
   -6652,  -6479,  -6317,  -6166,  -6022,  -5885,  -5756,  -5632,
   -5512,  -5340,  -5229,  -5121,  -5017,  -4913,  -4812,  -4719,
   -4627,  -4534,  -4446,  -4361,  -4278,  -4157,  -4082,  -4006,
   -3932,  -3862,  -3792,  -3723,  -3657,  -3591,  -3528,  -3464,
   -3402,  -3341,  -3250,  -3191,  -3133,  -3076,  -3018,  -2963,
   -2906,  -2851,  -2796,  -2742,  -2688,  -2635,  -2566,  -2522,
   -2478,  -2434,  -2389,  -2345,  -2301,  -2257,  -2212,  -2168,
   -2124,  -2080,  -2013,  -1969,  -1925,  -1880,  -1836,  -1792,
   -1748,  -1703,  -1659,  -1615,  -1571,  -1526,  -1482,  -1416,
   -1372,  -1327,  -1283,  -1239,  -1195,  -1156,  -1119,  -1082,
   -1044,  -1007,   -970,   -914,   -876,   -839,   -802,   -764,
    -727,   -690,   -652,   -615,   -578,   -541,   -503,   -447,
    -410,   -373,   -335,   -298,   -261,   -223,   -186,   -149,
    -111,    -74,    -37,      0
},
{ // quantization_level=248
  -32767, -13948, -12403, -11324, -10527,  -9926,  -9452,  -9061,
   -8723,  -8419,  -8139,  -7874,  -7619,  -7370,  -7125,  -6883,
   -6652,  -6479,  -6317,  -6166,  -6022,  -5885,  -5756,  -5632,
   -5512,  -5397,  -5284,  -5175,  -5069,  -4965,  -4862,  -4769,
   -4627,  -4534,  -4446,  -4361,  -4278,  -4196,  -4120,  -4043,
   -3969,  -3897,  -3827,  -3757,  -3690,  -3623,  -3559,  -3464,
   -3402,  -3341,  -3281,  -3220,  -3163,  -3104,  -3047,  -2990,
   -2935,  -2879,  -2824,  -2769,  -2715,  -2661,  -2611,  -2544,
   -2500,  -2456,  -2411,  -2367,  -2323,  -2279,  -2234,  -2190,
   -2146,  -2102,  -2057,  -2013,  -1969,  -1925,  -1858,  -1814,
   -1770,  -1726,  -1681,  -1637,  -1593,  -1549,  -1504,  -1460,
   -1416,  -1372,  -1327,  -1283,  -1239,  -1194,  -1138,  -1100,
   -1063,  -1026,   -988,   -951,   -914,   -876,   -839,   -802,
    -764,   -727,   -690,   -652,   -615,   -559,   -522,   -485,
    -447,   -410,   -373,   -335,   -298,   -261,   -223,   -186,
    -149,   -111,    -74,    -37,      0
},
{ // quantization_level=250
  -32767, -13948, -12403, -11324, -10527,  -9926,  -9452,  -9061,
   -8723,  -8419,  -8139,  -7874,  -7619,  -7370,  -7125,  -6883,
   -6745,  -6566,  -6398,  -6242,  -6094,  -5885,  -5756,  -5632,
   -5512,  -5397,  -5284,  -5175,  -5069,  -4965,  -4862,  -4762,
   -4671,  -4581,  -4490,  -4402,  -4319,  -4237,  -4157,  -4082,
   -4006,  -3932,  -3827,  -3757,  -3690,  -3623,  -3559,  -3495,
   -3433,  -3370,  -3311,  -3250,  -3191,  -3133,  -3076,  -3018,
   -2963,  -2906,  -2851,  -2796,  -2742,  -2688,  -2635,  -2566,
   -2522,  -2478,  -2434,  -2389,  -2345,  -2301,  -2257,  -2212,
   -2168,  -2124,  -2080,  -2035,  -1991,  -1947,  -1903,  -1858,
   -1814,  -1770,  -1726,  -1681,  -1615,  -1571,  -1526,  -1482,
   -1438,  -1394,  -1349,  -1305,  -1261,  -1217,  -1175,  -1138,
   -1100,  -1063,  -1026,   -988,   -951,   -914,   -876,   -839,
    -802,   -746,   -708,   -671,   -634,   -597,   -559,   -522,
    -485,   -447,   -410,   -373,   -335,   -298,   -261,   -223,
    -186,   -149,   -111,    -74,    -37,      0
},
{ // quantization_level=252
  -32767, -13948, -12403, -11324, -10527,  -9926,  -9452,  -9061,
   -8723,  -8419,  -8139,  -7874,  -7619,  -7370,  -7125,  -6883,
   -6745,  -6566,  -6398,  -6242,  -6094,  -5954,  -5820,  -5693,
```

Syntax

```
    -5571,  -5453,  -5340,  -5229,  -5121,  -5017,  -4913,  -4812,
    -4671,  -4581,  -4490,  -4402,  -4319,  -4237,  -4157,  -4082,
    -4006,  -3932,  -3862,  -3792,  -3723,  -3657,  -3591,  -3528,
    -3464,  -3402,  -3341,  -3281,  -3220,  -3163,  -3104,  -3047,
    -2990,  -2935,  -2879,  -2824,  -2769,  -2715,  -2661,  -2611,
    -2544,  -2500,  -2456,  -2411,  -2367,  -2323,  -2279,  -2234,
    -2190,  -2146,  -2102,  -2057,  -2013,  -1969,  -1925,  -1880,
    -1836,  -1792,  -1748,  -1703,  -1659,  -1615,  -1571,  -1526,
    -1482,  -1438,  -1394,  -1349,  -1305,  -1261,  -1217,  -1156,
    -1119,  -1082,  -1044,  -1007,   -970,   -932,   -895,   -858,
     -820,   -783,   -746,   -708,   -671,   -634,   -597,   -559,
     -522,   -485,   -447,   -410,   -373,   -335,   -298,   -261,
     -223,   -186,   -149,   -111,    -74,    -37,      0
  },
  { // quantization_level=254
    -32767, -13948, -12403, -11324, -10527,  -9926,  -9452,  -9061,
    -8723,  -8419,  -8139,  -7874,  -7619,  -7370,  -7125,  -6883,
    -6745,  -6566,  -6398,  -6242,  -6094,  -5954,  -5820,  -5693,
    -5571,  -5453,  -5340,  -5229,  -5121,  -5017,  -4913,  -4812,
    -4719,  -4627,  -4534,  -4446,  -4361,  -4278,  -4196,  -4120,
    -4043,  -3969,  -3897,  -3827,  -3757,  -3690,  -3623,  -3559,
    -3495,  -3433,  -3370,  -3311,  -3250,  -3191,  -3133,  -3076,
    -3018,  -2963,  -2906,  -2851,  -2796,  -2742,  -2688,  -2635,
    -2566,  -2522,  -2478,  -2434,  -2389,  -2345,  -2301,  -2257,
    -2212,  -2168,  -2124,  -2080,  -2035,  -1991,  -1947,  -1903,
    -1858,  -1814,  -1770,  -1726,  -1681,  -1637,  -1593,  -1549,
    -1504,  -1460,  -1416,  -1372,  -1327,  -1283,  -1239,  -1195,
    -1156,  -1119,  -1082,  -1044,  -1007,   -970,   -932,   -895,
     -858,   -820,   -783,   -746,   -708,   -671,   -634,   -597,
     -559,   -522,   -485,   -447,   -410,   -373,   -335,   -298,
     -261,   -223,   -186,   -149,   -111,    -74,    -37,      0
  },
  { // quantization_level=256
    -32767, -14975, -13114, -11819, -10892, -10206,  -9678,  -9249,
    -8886,  -8568,  -8278,  -8006,  -7746,  -7494,  -7248,  -7004,
    -6839,  -6652,  -6479,  -6317,  -6166,  -6022,  -5885,  -5756,
    -5632,  -5512,  -5397,  -5284,  -5175,  -5069,  -4965,  -4862,
    -4769,  -4671,  -4581,  -4490,  -4402,  -4319,  -4237,  -4157,
    -4082,  -4006,  -3932,  -3862,  -3792,  -3723,  -3657,  -3591,
    -3528,  -3464,  -3402,  -3341,  -3281,  -3220,  -3163,  -3104,
    -3047,  -2990,  -2935,  -2879,  -2824,  -2769,  -2715,  -2661,
    -2611,  -2566,  -2522,  -2478,  -2434,  -2389,  -2345,  -2301,
    -2257,  -2212,  -2168,  -2124,  -2080,  -2035,  -1991,  -1947,
    -1903,  -1858,  -1814,  -1770,  -1726,  -1681,  -1637,  -1593,
    -1549,  -1504,  -1460,  -1416,  -1372,  -1327,  -1283,  -1239,
    -1194,  -1156,  -1119,  -1082,  -1044,  -1007,   -970,   -932,
     -895,   -858,   -820,   -783,   -746,   -708,   -671,   -634,
     -597,   -559,   -522,   -485,   -447,   -410,   -373,   -335,
     -298,   -261,   -223,   -186,   -149,   -111,    -74,    -37,
    0
  }
};
```

10.8 LFE Constants and Tables

10.8.1 INTERPOLATION_FILTER_TABLE

Table 10-20: Interpolation_filter_table

Syntax

```
// INTERPOLATION_FILTER_TABLE

const float INTERPOLATION_FILTER_TABLE[512] =
{
  -7.668819067196072e-06, -7.212217984347475e-06,  9.113865273119975e-07,
   9.171480540931617e-07,  9.384579382365308e-07,  9.746923687020128e-07,
   1.025335256341165e-06,  1.089955906317944e-06,  1.168216337122893e-06,
   1.259848375880651e-06,  1.364646386311134e-06,  1.482478583482513e-06,
   1.613263794223289e-06,  1.756957644624211e-06,  1.913572286928453e-06,
   2.083162946421036e-06,  2.265805551604233e-06,  2.461633288450740e-06,
   2.670696816987063e-06,  2.893449436869689e-06,  3.129591447263766e-06,
```

Syntax
3.3796338381524599e-06, 3.6442037192188824e-06, 3.9227377476977048e-06, 4.2157353330332455e-06, 4.5238197617090697e-06, 4.8472191746119172e-06, 5.1859984978769941e-06, 5.5403682850554226e-06, 5.9107137383656799e-06, 6.2974548966859240e-06, 6.7009556102210311e-06, 7.1215063276937743e-06, 7.5593491811818040e-06, 8.0147509082905680e-06, 8.4880394769821444e-06, 8.9795502522425741e-06, 9.4896696473813119e-06, 1.0018676183886637e-05, 1.0566989135619957e-05, 1.1134832685332424e-05, 1.1722441104686407e-05, 1.2330271498912223e-05, 1.2958471998907036e-05, 1.3607303145803741e-05, 1.4277168644804541e-05, 1.4968377641429583e-05, 1.5681133666344951e-05, 1.6415679667583737e-05, 1.7172323907407854e-05, 1.7951336940336764e-05, 1.8752914565542560e-05, 1.9577237028484926e-05, 2.0424498987198162e-05, 2.1294910856083009e-05, 2.2188705119782307e-05, 2.3106042880478436e-05, 2.4047089198200457e-05, 2.5011934949357709e-05, 2.6000743656026869e-05, 2.7013661006796472e-05, 2.8050733623375222e-05, 2.9112115044919322e-05, 3.0197840572287782e-05, 3.1307886551968384e-05, 3.2442291550630757e-05, 3.3601091175026309e-05, 3.4784249756599108e-05, 3.5991713357821593e-05, 3.7223458934655719e-05, 3.8479436456339605e-05, 3.9759518821844634e-05, 4.1063558043649501e-05, 4.2391419809834134e-05, 4.3742940163446270e-05, 4.5117958585403426e-05, 4.6516239425583343e-05, 4.7937540234090846e-05, 4.9381578012749514e-05, 5.0848065717988151e-05, 5.2336719765934383e-05, 5.3847131552608536e-05, 5.5378921001939289e-05, 5.6931695578744350e-05, 5.8505003103095183e-05, 6.0098385520831817e-05, 6.1711379951898522e-05, 6.3343457754588188e-05, 6.4994025872993263e-05, 6.6662494568011009e-05, 6.8348280541511100e-05, 7.0050735592570059e-05, 7.1769158888016551e-05, 7.3502857990486413e-05, 7.5251076490795917e-05, 7.7013045485229843e-05, 7.8787938345954299e-05, 8.0574903277559485e-05, 8.2373073409184539e-05, 8.4181534990273395e-05, 8.5999375592861176e-05, 8.7825576676625838e-05, 8.9659094374458298e-05, 9.1498901618546232e-05, 9.3343926417056679e-05, 9.5193038785288723e-05, 9.7045089143355231e-05, 9.8898903458584137e-05, 1.0075324879311350e-04, 1.0260685578135060e-04, 1.0445845991190383e-04, 1.0630674131339569e-04, 1.0815028504143460e-04, 1.0998770862185373e-04, 1.1181758786720628e-04, 1.1363847818155884e-04, 1.1544889687842718e-04, 1.1724729778918734e-04, 1.1903212609217097e-04, 1.2080177235364789e-04, 1.2255463462643886e-04, 1.2428905075321346e-04, 1.2600331032450214e-04, 1.2769571483538231e-04, 1.2936454234373416e-04, 1.3100799190071533e-04, 1.3262422228279416e-04, 1.3421141376835909e-04, 1.3576773406174083e-04, 1.3729128043902213e-04, 1.3878013882552919e-04, 1.4023240331349397e-04, 1.4164604452180301e-04, 1.4301907871914579e-04, 1.4434952127328769e-04, 1.4563533673147788e-04, 1.4687448344910634e-04, 1.4806485598055409e-04, 1.4920437256871959e-04, 1.5029092440281441e-04, 1.5132240085515293e-04, 1.5229664795540548e-04, 1.5321145395723988e-04, 1.5406462732511206e-04, 1.5485402050488298e-04, 1.5557744937712029e-04, 1.5623266648984377e-04, 1.5681745072309989e-04, 1.5732963140961532e-04, 1.5776695302520839e-04, 1.5812709431327305e-04, 1.5840789951634669e-04, 1.5860711030349814e-04, 1.5872248696176537e-04, 1.5875185961929694e-04, 1.5869295612744606e-04, 1.5854353548214008e-04, 1.5830132829031433e-04, 1.5796413361383131e-04, 1.5752977106348918e-04, 1.5699605671554452e-04, 1.5636078394313819e-04, 1.5562172212664030e-04, 1.5477669868628959e-04, 1.5382365711394345e-04, 1.5276055554290255e-04, 1.5158524881694383e-04, 1.5029559755114561e-04, 1.4888967788330163e-04, 1.4736559984986311e-04, 1.4572113885280883e-04, 1.4395443928341502e-04, 1.4206362473728903e-04, 1.4004666552047570e-04, 1.3790181586353258e-04, 1.3562725784888846e-04, 1.3322129840391030e-04, 1.3068226508614370e-04, 1.2800852670344541e-04, 1.2519844430540240e-04, 1.2225035802755358e-04, 1.1916265557040345e-04, 1.1593384839402250e-04, 1.1256262793827424e-04, 1.0904785128965059e-04, 1.0538834904399469e-04, 1.0158267451769181e-04, 9.7629213184046896e-05, 9.3526948756359295e-05, 8.9275995660029753e-05, 8.4874397309115824e-05, 8.0321035385967532e-05, 7.5615943562724576e-05, 7.0757829115750611e-05, 6.5746229461680400e-05, 6.0580545629428425e-05, 5.5260301033096958e-05, 4.9785091494469531e-05, 4.4154607940022520e-05, 3.8368638489213868e-05, 3.2427047897601202e-05, 2.6329795677957754e-05, 2.0076949589750306e-05, 1.3668666396794861e-05, 7.1052004105248739e-06, 3.8692241041181390e-07, -6.4856695914500870e-06, -1.3511531993589918e-05, -2.0696269830824493e-05, -2.8022794532350482e-05, -3.5499063577323677e-05, -4.3139852298822455e-05, -5.0927179823178587e-05, -5.8859483152660485e-05, -6.6936657264134473e-05, -7.5158388871681900e-05, -8.3523849656844251e-05, -9.2031744727342783e-05, -1.0068041116125003e-04, -1.0946790215014821e-04, -1.1839208160353217e-04, -1.2745067353257640e-04, -1.3664126571117472e-04, -1.4596134017531295e-04, -1.5540829079751731e-04, -1.6497938389191483e-04, -1.7467184589147706e-04, -1.8448256087038749e-04, -1.9440878948530710e-04, -2.0444707202473057e-04, -2.1459395083182485e-04, -2.2484654470799256e-04, -2.3520092750030068e-04, -2.4565320254653692e-04, -2.5619971736427435e-04, -2.6683665938218476e-04, -2.7755986349580505e-04, -2.8836495536267839e-04, -2.9924751942454107e-04, -3.1020309973454619e-04, -3.2122712852716145e-04, -3.3231487404765007e-04, -3.4346140547654647e-04, -3.5466162507653842e-04, -3.6591035052959168e-04, -3.7720226888196702e-04, -3.8853203397227771e-04, -3.9989404776117580e-04, -4.1128272560662753e-04, -4.2269228058227558e-04, -4.3411664149667394e-04, -4.4554988479862792e-04,

Syntax
-4.5698581460158516e-04, -4.6841804293440003e-04, -4.7984019532145935e-04,
-4.9124582888784819e-04, -5.0262829436669408e-04, -5.1398077680173734e-04,
-5.2529640389896166e-04, -5.3656823220735454e-04, -5.4778917335388357e-04,
-5.5895200129910041e-04, -5.7004938513818197e-04, -5.8107389523457743e-04,
-5.9201806846455437e-04, -6.0287430830400569e-04, -6.1363494811268813e-04,
-6.2429214550797278e-04, -6.3483799930820961e-04, -6.4526459059212435e-04,
-6.5556380345782941e-04, -6.6572753703446081e-04, -6.7574760503497674e-04,
-6.8561566112126352e-04, -6.9532330596543590e-04, -7.0486213284649065e-04,
-7.1422365673939957e-04, -7.2339928960864057e-04, -7.3238040884536688e-04,
-7.4115837528616663e-04, -7.4972446318269197e-04, -7.5806986064742312e-04,
-7.6618573588285049e-04, -7.7406320435754095e-04, -7.8169339347783622e-04,
-7.8906736510879992e-04, -7.9617615542074062e-04, -8.0301075755773328e-04,
-8.0956213407429919e-04, -8.1582131263904518e-04, -8.2177923064644153e-04,
-8.2742681510307719e-04, -8.3275502877523417e-04, -8.3775481852983908e-04,
-8.4241712842907980e-04, -8.4673294771058958e-04, -8.5069328643105392e-04,
-8.5428912409305803e-04, -8.5751144688439400e-04, -8.6035132142028240e-04,
-8.6279985628620907e-04, -8.6484815122332042e-04, -8.6648738428143305e-04,
-8.6770875537008324e-04, -8.6850353727354755e-04, -8.6886304947703780e-04,
-8.6877866155300179e-04, -8.6824183378681429e-04, -8.6724407520960339e-04,
-8.6577703278012776e-04, -8.6383239621168958e-04, -8.6140189784679726e-04,
-8.5847739792044883e-04, -8.5505088300202674e-04, -8.5111442169139582e-04,
-8.4666018077612112e-04, -8.4168046127913732e-04, -8.3616767372544562e-04,
-8.3011431587853596e-04, -8.2351304510212321e-04, -8.1635669104741760e-04,
-8.0863813626249476e-04, -8.0035043782472474e-04, -7.9148680681414684e-04,
-7.8204062014736295e-04, -7.7200543479467735e-04, -7.6137492841046102e-04,
-7.5014296026209161e-04, -7.3830352669097842e-04, -7.2585083833671634e-04,
-7.1277929244321729e-04, -6.9908342999516315e-04, -6.8475799470934726e-04,
-6.6979797128239856e-04, -6.5419852630086242e-04, -6.3795497464263335e-04,
-6.2106285379004867e-04, -6.0351797218691900e-04, -5.8531632581443002e-04,
-5.6645409651035318e-04, -5.4692777744775678e-04, -5.2673398772119350e-04,
-5.0586958185528438e-04, -4.8433170393021729e-04, -4.6211771891801920e-04,
-4.3922526949416137e-04, -4.1565218824039752e-04, -3.9139657490660677e-04,
-3.6645677733409068e-04, -3.4083142688227594e-04, -3.1451942595675032e-04,
-2.8751988195637720e-04, -2.5983213969352567e-04, -2.3145587956040074e-04,
-2.0239110098150279e-04, -1.7263801481475107e-04, -1.4219706272351735e-04,
-1.1106905976869218e-04, -7.9255114437390851e-05, -4.6756428794342506e-05,
-1.3574655438833959e-05, 2.0288291724904981e-05, 5.4830269258308692e-05,
9.0048727428332497e-05, 1.2594090861898591e-04, 1.6250375569340248e-04,
1.9973398053336265e-04, 2.3762797041540178e-04, 2.7618177152581591e-04,
3.1539110495974303e-04, 3.5525140053549768e-04, 3.9575786130966400e-04,
4.3690544158500697e-04, 4.7868870633069229e-04, 5.2110179574842272e-04,
5.6413862801125818e-04, 6.0779304328350550e-04, 6.5205849006757437e-04,
6.9692776244935232e-04, 7.4239389313257977e-04, 7.8844946348206239e-04,
8.3508654972483691e-04, 8.8229725079657490e-04, 9.3007314424508388e-04,
9.7840557905124153e-04, 1.0272855480938778e-03, 1.0767037571103636e-03,
1.1266506247002157e-03, 1.1771163228858168e-03, 1.2280908435787032e-03,
1.2795639906937010e-03, 1.3315253067303957e-03, 1.3839639562836941e-03,
1.4368685759627497e-03, 1.4902272854200910e-03, 1.5440281208183094e-03,
1.5982597863054481e-03, 1.6529115426441514e-03, 1.7079686038501977e-03,
1.7634184241372914e-03, 1.8192496307660915e-03, 1.8754477643797584e-03,
1.9319998259463909e-03, 1.9888920947001743e-03, 2.0461106917061276e-03,
2.1036415038267328e-03, 2.1614702098034927e-03, 2.2195822790479279e-03,
2.2779629585090748e-03, 2.3365972996386847e-03, 2.3954701678854792e-03,
2.4545662243975730e-03, 2.5138699432796399e-03, 2.5733656277462879e-03,
2.6330373983874888e-03, 2.6928693512188202e-03, 2.7528440441556348e-03,
2.8129485953089428e-03, 2.8731632078845791e-03, 2.9334690448814317e-03,
2.9938548051192350e-03, 3.0543022319813487e-03, 3.1147928893109017e-03,
3.1753091405863601e-03, 3.2358338636813743e-03, 3.2963500180074414e-03,
3.3568404021611827e-03, 3.4172875818357408e-03, 3.4776738889887126e-03,
3.5379814890697425e-03, 3.5981924641822886e-03, 3.6582888373027575e-03,
3.7182525949373008e-03, 3.7780657349016935e-03, 3.8377102001669831e-03,
3.8971680646459539e-03, 3.9564210658774683e-03, 4.0154513646048079e-03,
4.0742408987959871e-03, 4.1327712216039239e-03, 4.1910245975328555e-03,
4.2489829784871100e-03, 4.3066280810148258e-03, 4.3639418103482708e-03,
4.4209062798019599e-03, 4.4775035956643337e-03, 4.5337157849602181e-03,
4.5895249082956134e-03, 4.6449131521389452e-03, 4.6998628335777502e-03,
4.7543563811806751e-03, 4.8083762873203191e-03, 4.8619050747417229e-03,
4.9149253917405364e-03, 4.9674199648710796e-03, 5.0193717669871421e-03,
5.0707637895068295e-03, 5.1215792350425655e-03, 5.1718015194193905e-03,
5.2214139713394952e-03, 5.2704002450961089e-03, 5.3187441828982263e-03,
5.3664296844896020e-03, 5.4134408483963517e-03, 5.4597620599348912e-03,
5.5053779001124530e-03, 5.5502730735109383e-03, 5.5944324792275863e-03,
5.6378412823975214e-03, 5.6804848805461893e-03, 5.7223488792450955e-03,
5.7634191148330646e-03, 5.8036816301094392e-03, 5.8431227790947853e-03,
5.8817291553337550e-03, 5.9194876622768906e-03, 5.9563854108178084e-03,
5.9924097379417804e-03, 6.0275483657287370e-03, 6.0617892122536302e-03,
6.0951205162044202e-03, 6.1275308709657144e-03, 6.1590091213768764e-03,

Syntax

```
   6.1895443737827955e-03,  6.2191260867996909e-03,  6.2477440744167735e-03,
   6.2753884405690793e-03,  6.3020495918311541e-03,  6.3277183059215810e-03,
   6.3523857048689786e-03,  6.3760432029354008e-03,  6.3986825698880961e-03,
   6.4202958921424653e-03,  6.4408756544770867e-03,  6.4604146929444937e-03,
   6.4789062055311672e-03,  6.4963437467322376e-03,  6.5127211761957550e-03,
   6.5280328067088608e-03,  6.5422733010129224e-03,  6.5554376525278762e-03,
   6.5675212522800447e-03,  6.5785198767057086e-03,  6.5884296569655894e-03,
   6.5972471138082169e-03,  6.6049691875606064e-03,  6.6115931864430648e-03,
   6.6171167464533403e-03,  6.6215379058877505e-03,  6.6248551411808229e-03,
   6.6270672735935518e-03,  6.6281735388554848e-03
};
```

Annex A (informative): Bibliography

ETSI EN 300 468: "Digital Video Broadcasting (DVB); Specification for Service Information (SI) in DVB systems".

ETSI TS 101 154: "Digital Video Broadcasting (DVB); Specification for the use of Video and Audio Coding in Broadcasting Applications based on the MPEG-2 Transport Stream".

Vesa Valimaki: "Discrete-time modeling of acoustic tubes using fractional delay filters", Doctoral Thesis. chap 3, p117, 1995.

Y.Qian et al.: "Pseudo-multi-tap filters in a low bit-rate CELP speech coder", Speech Communication, 1994.

"Adaptive Transform Coder Having Long Term Predictor", Patent 5,012,517, 1991.

Annex B (normative): DTS-UHD ISOBMFF File Encapsulation

B.1 Overview

In this clause, operational rules for boxes defined in ISO/IEC 14496-12 [4], as well as definitions of private extensions to those ISO file format standards, are described.

An ISO media file may contain one or more audio tracks. The tracks are composed in conformity to ISO base media file format and described in ISO/IEC 14496-12 [4], for an audio track structure. The clauses that follow describe the construction of an audio track containing DTS-UHD audio.

B.2 Signalling

B.2.1 Track Header

DTS-UHD tracks shall be compliant with audio tracks as defined in ISO/IEC 14496-12 [4]. The following rules shall be applied to the boxes within the Media Box in a DTS-UHD track:

- The `handler_type` field in the Handler Reference Box shall be set to `'soun'`.
- The Media Information Header Box shall be of type `'smhd'`.
- The Sample Description Box shall be derived from AudioSampleEntry as described in clause B.2.2.2.
- The `timescale` parameter in the Media Header Box shall be set to `samplerate` or an integer multiple thereof. Configuration of `samplerate` is described in clause B.2.2.2.
- DTS-UHD samples may be either sync samples or non-sync samples, so a Sync Sample Box may be present.

B.2.2 Sample Description Box

B.2.2.1 Overview of Sample Description Box

The SampleEntry box for DTS-UHD is derived from the AudioSampleEntry box defined in ISO/IEC 14496-12 [4]. The DTS-UHD specific SampleEntry box is identified by a unique codingname value (see Table B-1), indicating the audio format and profile used to encode the audio track.

Table B-1: Defined Audio Formats

codingname	Description
dtsx	DTS-UHD bitstream compatible with Decoder Profile = 2
dtsy	DTS-UHD bitstream compatible with Decoder Profile = 3

B.2.2.2 DTSUHDSampleEntry

`DTSUHDSampleEntry` extends the `AudioSampleEntry` box defined in ISO/IEC 14496-12 [4].

```
class DTSUHDSampleEntry (codingname) extends AudioSampleEntry (codingname) {
    DTSUHDSpecificBox() // 'udts' box
}
```

For `DTSUHDSampleEntry`(), the following values inherited from `AudioSampleEntry` are set as follows:

codingname shall be set to one of the values listed in Table B-1.

channelcount shall be set to the number of decodable output waveforms.

samplesize shall be set to 16.

samplerate shall be set to the base sampling frequency, as indicated by the *BaseSamplingFreqCode* field of the `DTSUHDSpecificBox`. For example, if *BaseSamplingFrequency* is 1, then *samplerate* is 48 000.

B.2.2.3 DTSUHDSpecificBox

B.2.2.3.1 Syntax of DTSUHDSpecificBox

The syntax and semantics of the DTSUHDSpecificBox (`'udts'`) are shown below.

Table B-2: DTS-UHD Specific Box

```
Syntax
class DTSUHDSpecificBox extends Box ('udts') {
    bit (6)                 DecoderProfileCode;
    bit (2)                 FrameDurationCode;
    bit (3)                 MaxPayloadCode;
    bit (5)                 NumPresentationsCode;
    unsigned int(32)        ChannelMask;
    bit (1)                 BaseSamplingFrequencyCode
    bit (2)                 SampleRateMod
    bit (3)                 RepresentationType;
    bit (3)                 StreamIndex
    bit (1)                 ExpansionBoxPresent;
    bit (NumPresentations)  IDTagPresent[NumPresentations];
    ByteAlign [0..7]
    for (i = 0; i < NumPresentations; i++){
        if (IDTagPresent[i] == TRUE)
            unsigned char   PresentationIDTag[16];
    }
    if (ExpansionBoxPresent) {
        box DTSExpansionBox [];
    }
}
```

B.2.2.3.2 Semantics for DTSUHDSpecificBox

DecoderProfileCode

DecoderProfile indicates the bitstream profile. *DecoderProfile* = *DecoderProfileCode* + 2. *DecoderProfile* indicates the decoder version required to fully process the encapsulated elementary stream.

FrameDurationCode

FrameDurationCode indicates *FrameDuration* according to Table B-3. The units for *FrameDuration* are sample periods relative to *BaseSamplingFrequency*, (see Table B-6).

Table B-3: Frame Duration

FrameDurationCode	FrameDuration (samples)
0	512
1	1 024
2	2 048
3	4 096

MaxPayloadCode

MaxPayloadCode indicates the maximum size of the audio payload, as indicated in Table B-4. Note that *MaxPayload* is not the size of the largest audio frame in the presentation, but rather a "not to exceed" value for buffer configuration and digital audio interface purposes, and is inclusive of all required preambles, headers, burst spacing, etc.

Table B-4: MaxPayload

MaxPayloadCode	MaxPayload (bytes)
0	2 048
1	4 096
2	8 192
3	16 384
4	32 768
5	65 536
6	131 072
7	reserved

Note that *MaxPayloadCode* can be used to determine the minimum ISO/IEC 60958 [10] frame rate required to transfer the DTS-UHD stream using a digital audio interface.

NumPresentationsCode

NumPresentations corresponds to the parameter *m_ucNumAudioPres* extracted from the bitstream and represents the number of audio presentations encoded within DTS-UHD elementary stream. The valid range of the bitstream parameter *m_ucNumAudioPres* is from 1 to 32. The relationship between *NumPresentations*, *NumPresentationsCode* and *m_ucNumAudioPres* shall be as follows:

$$m_ucNumAudioPres = NumPresentations = NumPresentationsCode + 1$$

ChannelMask

A bit mask that indicates the channel layout encoded in the default presentation of the DTS-UHD bitstream according to the bit-to-channel-label mapping described in Table B-5. Note that *ChannelMask* table is derived from Channel Bitmask defined in ETSI TS 103 584 [7].

Table B-5: Speaker Labels for ChannelMask

Label	Notation	ChannelMask
C	CENTER	0x00000001
L	LEFT	0x00000002
R	RIGHT	0x00000004
Ls	SRRD_LEFT	0x00000008
Rs	SRRD_RIGHT	0x00000010
LFE1	LFE_1	0x00000020
Cs	SRRD_CENTER	0x00000040
Lsr	REAR_SRRD_LEFT	0x00000080
Rsr	REAR_SRRD_RIGHT	0x00000100
Lss	SIDE_SRRD_LEFT	0x00000200
Rss	SIDE_SRRD_RIGHT	0x00000400
Lc	LEFT_CENTER	0x00000800
Rc	RIGHT_CENTER	0x00001000
Lh	HIGH_LEFT	0x00002000
Ch	HIGH_CENTER	0x00004000
Rh	HIGH_RIGHT	0x00008000
LFE2	LFE_2	0x00010000
Lw	LEFT_WIDE	0x00020000
Rw	RIGHT_WIDE	0x00040000
Oh	TOP_CENTER_SRRD	0x00080000
Lhs	HIGH_SIDE_LEFT	0x00100000
Rhs	HIGH_SIDE_RIGHT	0x00200000
Chr	HIGH_REAR_CENTER	0x00400000
Lhr	HIGH_REAR_LEFT	0x00800000
Rhr	HIGH_REAR_RIGHT	0x01000000
Cb	LOW_FRONT_CENTER	0x02000000
Lb	LOW_FRONT_LEFT	0x04000000
Rb	LOW_FRONT_RIGHT	0x08000000
Ltf	TOP_FRONT_LEFT	0x10000000
Rtf	TOP_FRONT_RIGHT	0x20000000
Ltr	TOP_REAR_LEFT	0x40000000
Rtr	TOP_REAR_RIGHT	0x80000000

BaseSamplingFrequencyCode

BaseSamplingFrequencyCode is derived from the parameter *m_unClockRateInHz* defined in the present document. This field shall indicate the base sampling frequency, according to Table B-6.

Table B-6: Base Sampling Frequency

BaseSamplingFrequencyCode	Clock Rate (Hz)
0	44 100
1	48 000

SampleRateMod

This field indicates the sampling frequency of the audio samples stored in the bitstream. It modifies the *BaseSamplingFrequency* by multiplying by the corresponding value indicated in Table B-7.

Table B-7: SampleRateMod

SampleRateMod	Multiplier
0	1
1	2
2	4
3	8

RepresentationType

This value conveys general information about the type of spatial audio signals encoded in the default presentation of the DTS-UHD bit-stream according to the description in Table B-8. This parameter shall match the bitstream metadata parameter *ucObjRepresTypeIndex* for the default presentation in the stream.

Allowed values for *RepresentationType* are affected by, and may affect, *ChannelMask* as indicated.

Table B-8: Representation Type

RepresentationType	Description
0	Encoded channel representation is described by *ChannelMask*
1	Multi-channel representation in layout described by *ChannelMask* obtained by rendering 2D content (i.e. no height channels), with spatial resolution higher than indicated by the encoded layout (e.g. LtRt, 5.1ES)
2	Multi-channel representation in layout described by *ChannelMask* obtained by rendering 3D content (i.e. includes height), with spatial resolution higher than indicated by the encoded layout (e.g. 5.1 NeoX)
3	Binaurally processed audio (2 waveforms, L, R for headphones)
4	Ambisonic representation
5	Audio tracks with or without an associated mixing matrix to a particular channel mask based output layout
6	3D Objects based representation
7	A combination of 3D sound objects, channels and/or sound fields

When *RepresentationType* has a value of 1 or 2, *ChannelMask* shall be set to a value that corresponds to the layout of the encoded channels into which higher spatial resolution components have been rendered (i.e. the corresponding channels that have been delivered to the audio compression engine). For example, LtRt material would have a *ChannelMask* of 0x00000006, representing the two channels of encoded audio content that may be presented at a higher spatial resolution on, for example, a 5.1-channel system.

When *RepresentationType* is equal to 3, *ChannelMask* shall be set to 0x00000006, indicating Left and Right channels are active.

When *RepresentationType* is greater than 3, *ChannelMask* shall be set to 0x00000000.

StreamIndex

When a given audio program is delivered using multiple streams, StreamIndex can be used to indicate stream priority. The main stream shall have *StreamIndex*=0, while auxiliary streams shall have *StreamIndex* values in the range of 1-7. Note that some use cases using DASH delivery will not depend on *StreamIndex*, but rather rely on the *DependencyId* parameter in the DASH MPD to orchestrate playback.

If the Audio Program is limited to a single stream, then StreamIndex shall be set to 0.

ExpansionBoxPresent

This flag, if set to 1, indicates that at least one private box is present at the end of the DTSUHDSpecificBox.

IDTagPresent

One *IDTagPresent* flag exists for each presentation in the audio track, with the total number indicated by *NumPresentations*. The *IDTagPresent* flags appear in order of the increasing value of the Audio Presentation Index (*ucAudPresIndex*). If *IDTagPresent* is equal to 1 for a given presentation, then a 16-byte ID for that presentation (*PresentationIDTag*) shall be present in the descriptor.

ByteAlign

A variable number of bits (from 0 to 7) shall be inserted to pad to the end of the current byte boundary. See the note regarding ByteAlign bits in clause 3.4.

PresentationIDTag

PresentationIDTag for a specific audio presentation shall be present only when the corresponding *IDTagPresent* flag is set to 1. This 16-byte field can be used to identify the corresponding audio presentation.

B.2.2.3.3 DTSExpansionBox

If *ExpansionBoxPresent* is set to 1, then one or more private boxes referred to as *DTSExpansionBoxes* can be found at the end of the 'udts' box, filling the remaining space of the box. As defined in ISO/IEC 14496-12 [4] boxes with an unrecognized type shall be ignored by the receiver.

Playback devices not equipped to support one or more of these private boxes shall be able to depend on the defined parameters in 'udts' to play the audio track with its default presentation parameters.

B.3 Storage of DTS-UHD Elementary Streams

B.3.1 DTS-UHD Samples

Storage of DTS-UHD elementary streams within an ISO media file shall be according to this clause.

A sample, as defined in ISO/IEC 14496-12 [4] shall consist of a single DTS-UHD audio frame, including the leading syncword, as defined in the present document.

B.3.2 Random Access Points

A DTS-UHD audio sample may be either a sync frame or non-sync frame, as identified by the first four bytes of the sample, which constitute the SYNCWORD, shown in Table B-9.

Table B-9: DTS-UHD SYNCWORD

SYNCWORD	Designation
0x40411BF2	DTS-UHD Sync Frame
0x71C442E8	DTS-UHD Non-sync Frame

For all DTS-UHD tracks, the first sample in the track shall be a sync frame. For fragmented track files, the first sample in each fragment shall be a sync frame.

For track fragments containing multiple mdat boxes, the first DTS-UHD frame stored in each mdat shall be a sync frame.

For DTS-UHD tracks containing non-sync frames, the methods provided in ISO/IEC 14496-12 [4] for locating random access points such as the sync sample box ('stss'), the track fragment random access box ('tfra'), and random access information in 'trun', 'traf' and 'trex', shall be applied as required.

Annex C (normative): DTS-UHD in MPEG-2 Transport Streams

C.1 Introduction

This clause defines the signalling and encapsulation of DTS-UHD bitstreams, as defined in the present document, for use in MPEG-2 Transport Streams in compliance with ISO/IEC 13818-1 [8].

C.2 MPEG-2 System Signalling

C.2.1 Buffering Model

It is recommended that the main audio buffer size (BS_n, as defined in ISO/IEC 13818-1 [8]) has a fixed value of 66 434 bytes.

C.2.2 DTS-UHD Audio Descriptor

C.2.2.1 Descriptor Placement

The DTS-UHD Audio Descriptor shall be included in the program map section following the relevant ES_info_length field for any DTS-UHD audio stream that is included in the MPEG program.

C.2.2.2 Descriptor Syntax

The syntax for the DTS-UHD Audio Descriptor shall be as defined in Table C-1. All values presented in the DTS-UHD descriptor are interpreted as unsigned integer values, most significant bit first.

Table C-1: DTS-UHD descriptor

Syntax	size (bits)	mnemonic
dts-uhd_descriptor {		
descriptor_tag	8	uimsbf
descriptor_length	8	uimsbf
if (descriptor_tag = 0x7F) descriptor_tag_extension (8)	8	uimsbf
DecoderProfileCode	6	uimsbf
FrameDurationCode	2	bslbf
MaxPayloadCode	3	bslbf
ExtendedDescriptor	1	bslbf
LongDescriptor	1	bslbf
StreamIndex	3	bslbf
if (LongDescriptor) {		
NumPresentationsCode	5	uimsbf
ChannelMask	32	bslbf
BaseSamplingFreqCode	1	bslbf
SampleRateMod	2	bslbf
RepresentationType	3	bslbf
for (np=0; np<NumPresentations; np++) {		
IDTagPresent[np]	NumPresentations	uimsbf
}		
ByteAlign	0-7	bslbf
for (np=0; np<NumPresentations; np++) {		
if (IDTagPresent[np]) {		
PresentationIDTag[np]	128	
} // end of If IDTagPresent		
} // end of ID tag loop		

Syntax	size (bits)	mnemonic

} // end of if LongDescriptor
if (ExtendedDescriptor) {
ByteCount
 	
reserved_bits
ExtendedPayloadBytes[ByteCount]
}
}
	
	6
	2
	ByteCount * 8
		
		uimsbf
		bslbf
		bslbf

C.2.2.3 Semantics for Parameters

descriptor_tag

The value of *descriptor_tag* shall be 0x7F and the *value of descriptor_tag_extension* shall be set to 0x21, as assigned for use in DVB, ATSC and SCTE.

DecoderProfileCode

DecoderProfile indicates the DTS-UHD decoder profile required to decode this stream. Valid range of *DecoderProfileCode* shall be from 0 to 63. The relationship between the *DecoderProfile* and *DecoderProfileCode* shall be as follows:

DecoderProfile = DecoderProfileCode + 2

where *DecoderProfile* indicates the DTS-UHD decoder profile as defined in Annex F.

FrameDurationCode

The frame duration code indicates *FrameDuration* in units of audio PCM samples at the base sampling frequency, according to Table C-2. The *FrameDuration* in seconds is *FrameDuration* divided by the Clock Rate from Table C-5.

Table C-2: Frame Duration

FrameDurationCode	FrameDuration (samples)
0	512
1	1 024
2	2 048
3	4 096

MaxPayloadCode

MaxPayloadCode indicates the maximum size of the audio payload, as indicated in Table C-3. Note that *MaxPayload* is not the size of the largest audio frame in the presentation, but rather a "not to exceed" value for buffer configuration and digital audio interface purposes, and is inclusive of all required preambles, headers, burst spacing, etc.

Table C-3: MaxPayload

MaxPayloadCode	MaxPayload (bytes)
0	2 048
1	4 096
2	8 192
3	16 384
4	32 768
5	65 536
6	131 072
7	reserved

Note that *MaxPayloadCode* can be used to determine the minimum ISO/IEC 60958 [10] frame rate required to transfer the DTS-UHD stream using a digital audio interface.

ExtendedDescriptor

ExtendedDescriptor is a 1-bit flag to be interpreted as a Boolean. If *ExtendedDescriptor* == 1, then additional private information bytes beyond this descriptor definition shall exist at the end of this definition. *ExtendedPayloadBytes* shall be included in *descriptor_length*.

LongDescriptor

LongDescriptor is a 1-bit flag to be interpreted as a Boolean. When *LongDescriptor* is set to 1 the *LongDescriptor* parameters shall be correctly populated.

StreamIndex

When an audio program is delivered using multiple streams, *StreamIndex* may be used to indicate stream priority for prioritizing mixing metadata. The main stream shall have *StreamIndex*=0, while auxiliary streams shall have *StreamIndex* values in the range of 1-7. Note that some use cases using DASH delivery will not depend on *StreamIndex*, but rather rely on *DependencyId* in the MPD to orchestrate playback.

If the audio program is delivered in a single stream, *StreamIndex* shall be 0.

reserved_bits

These bits serve to byte align some parameters in the descriptor and may be defined for use in future specifications. For the purposes of the present document, all *reserved_bits* shall be set to 0 and ignored by the receiver.

ExtendedPayloadBytes

The number of bytes, *ByteCount*, is determined by parsing the parameters defined here and comparing the number of bytes remaining to be parsed based on *descriptor_length*. If *ExtendedPayloadBytes* exist, receivers built to the present document shall ignore this data.

NumPresentationsCode

This field represents the number of audio presentations encoded within DTS-UHD elementary stream. The valid range of the bitstream parameter *m_ucNumAudioPres* is from 1 to 32. The relationship between *NumPresentations*, *NumPresentationsCode* and *m_ucNumAudioPres* shall be as follows:

m_ucNumAudioPres = NumPresentations = NumPresentationsCode + 1.

ChannelMask

This parameter is only present when *LongDescriptor* is equal to 1. A bit-mask that indicates the channel layout encoded in the default presentation of the DTS-UHD bitstream according to the bit-to-channel-label mapping described in Table C-4. Note that the ChannelMask table is derived from Channel Bitmask defined in ETSI TS 103 584 [7].

Table C-4: Speaker Labels for ChannelMask

Label	Notation	ChannelMask
C	CENTER	0x00000001
L	LEFT	0x00000002
R	RIGHT	0x00000004
Ls	SRRD_LEFT	0x00000008
Rs	SRRD_RIGHT	0x00000010
LFE1	LFE_1	0x00000020
Cs	SRRD_CENTER	0x00000040
Lsr	REAR_SRRD_LEFT	0x00000080
Rsr	REAR_SRRD_RIGHT	0x00000100
Lss	SIDE_SRRD_LEFT	0x00000200
Rss	SIDE_SRRD_RIGHT	0x00000400
Lc	LEFT_CENTER	0x00000800
Rc	RIGHT_CENTER	0x00001000
Lh	HIGH_LEFT	0x00002000
Ch	HIGH_CENTER	0x00004000
Rh	HIGH_RIGHT	0x00008000
LFE2	LFE_2	0x00010000
Lw	LEFT_WIDE	0x00020000
Rw	RIGHT_WIDE	0x00040000
Oh	TOP_CENTER_SRRD	0x00080000
Lhs	HIGH_SIDE_LEFT	0x00100000
Rhs	HIGH_SIDE_RIGHT	0x00200000
Chr	HIGH_REAR_CENTER	0x00400000
Lhr	HIGH_REAR_LEFT	0x00800000
Rhr	HIGH_REAR_RIGHT	0x01000000
Cb	LOW_FRONT_CENTER	0x02000000
Lb	LOW_FRONT_LEFT	0x04000000
Rb	LOW_FRONT_RIGHT	0x08000000
Ltf	TOP_FRONT_LEFT	0x10000000
Rtf	TOP_FRONT_RIGHT	0x20000000
Ltr	TOP_REAR_LEFT	0x40000000
Rtr	TOP_REAR_RIGHT	0x80000000

BaseSamplingFrequencyCode

This parameter is only present when *LongDescriptor* is equal to 1. *BaseSamplingFrequencyCode* is derived from the parameter *m_unClockRateInHz* defined in the bitstream. This field shall indicate the base sampling frequency, according to Table C-5.

Table C-5: Base Sampling Frequency

BaseSamplingFrequencyCode	Clock Rate (Hz)
0	44 100
1	48 000

SampleRateMod

This parameter is only present when *LongDescriptor* is equal to 1. This field indicates the sampling frequency of the audio samples stored in the bitstream. It modifies the *BaseSamplingFrequency* by multiplying by the corresponding value indicated in Table C-6.

Table C-6: SampleRateMod

SampleRateMod	Multiplier
0	1
1	2
2	4
3	8

RepresentationType

This parameter is only present when *LongDescriptor* is equal to 1. This value conveys general information about the type of spatial audio components encoded in the default presentation of the DTS-UHD bit-stream according to the description in Table C-7. This parameter shall match the bitstream metadata parameter *ucObjRepresTypeIndex* defined in the bitstream.

Allowed values for *RepresentationType* are affected by and may affect *ChannelMask*, as indicated.

Table C-7: Representation Type

RepresentationType	Description
0	Multi-channel representation in layout described by *ChannelMask*
1	Multi-channel representation in layout described by *ChannelMask* obtained by rendering 2D content, (i.e. no height channels), with spatial resolution higher than indicated by the encoded layout (e.g. LtRt, 5.1ES)
2	Multi-channel representation in layout described by *ChannelMask* obtained by rendering 3D content (i.e. includes height), with spatial resolution higher than indicated by the encoded layout (e.g. 5.1 NeoX)
3	Binaurally processed audio (2 waveforms, L, R for headphones)
4	Ambisonic representation
5	Audio tracks with or without an associated mixing matrix to a particular channel mask based output layout
6	3D Objects based representation
7	Combination of components with different representation types

When *RepresentationType* has a value of 0, *ChannelMask* shall be set to a value that directly corresponds to the encoded channels.

When *RepresentationType* has a value of 1 or 2, *ChannelMask* shall be set to a value that corresponds to the layout of the encoded channels into which higher spatial resolution components have been rendered (i.e. the corresponding channels that have been delivered to the audio compression engine). For example, LtRt material would have a *ChannelMask* of 0x00000006, representing the two channels of encoded audio content that may be presented at a higher spatial resolution on, for example, a 5.1-channel system.

When *RepresentationType* is equal to 3, *ChannelMask* shall be set to 0x00000006, indicating Left and Right channels are active.

When *RepresentationType* is greater *than* 3, *ChannelMask* shall be set to 0x00000000, indicating that the audio presentation is channel layout agnostic.

IDTagPresent

This parameter is only present when *LongDescriptor* is equal to 1. One *IDTagPresent* flag exists for each presentation in the audio track, with the total number indicated by *NumPresentations*. The *IDTagPresent* flags appear in order of the increasing value of the Audio Presentation Index (*ucAudPresIndex*). If *IDTagPresent* is equal to *1* for a given presentation, then a 16-byte ID for that presentation (*PresentationIDTag*) shall be present in the descriptor.

ByteAlign

A variable number of bits (from 0 to 7) shall be inserted to pad to a byte boundary. See the note regarding ByteAlign bits in clause 3.4.

PresentationIDTag

This parameter is only present when *LongDescriptor* is equal to 1. *PresentationIDTag* for a specific audio presentation shall be present only when the corresponding *IDTagPresent* flag is set to 1. This 16-byte field can be used to identify the corresponding audio presentation.

C.3 DTS-UHD PES Packet Encapsulation

C.3.1 Stream Type

For DTS-UHD transport streams, *stream_type* shall be set to 0x06 indicating stream is carrying a private data type in PES packets. The mapping of values of PID to *stream_type* is indicated in the transport stream PMT.

C.3.2 Stream ID

All DTS-UHD elementary streams shall use a *stream_id* of 0xBD, indicating private stream 1, in accordance with ISO/IEC 13818-1 [8]. Multiple DTS streams may share the same value of *stream_id* since each stream is carried with a unique PID value.

C.3.3 Audio Access Unit Alignment in PES Packets

DTS-UHD access units shall be delivered in the MPEG-TS in playback order. Multiple access units are permitted in a PES packet. Audio access units shall begin with a valid DTS sync word. Valid DTS sync words are listed in Table C-8.

Table C-8: DTS-UHD Syncwords

Name	Syncword	Description
DTSUHD_SYNC	0x40411BF2	DTS-UHD Sync Frame
DTSUHD_NOSYNC	0x71C442E8	DTS-UHD Non-sync Frame

C.3.4 Random Access Points

If a PES packet is to be used for random access to a DTS-UHD stream the following conditions shall be satisfied according to ISO/IEC 13818-1 [8]:

- Optional PES header shall be present and shall contain PTS.
- *Data_Alignment_Indicator* within the PES header shall be set to 1.
- The first access unit in the PES packet shall be a DTS-UHD sync frame, as indicated by the presence of DTSUHD_SYNC.

The transport packet that contains the header of a random-access PES packet may have the *random_access_indicator*, if present, set to 1. Transport packets that do not meet this requirement shall have the *random_access_indicator*, if present, set to 0.

Annex D (normative): Delivery of DTS-UHD using MPEG-DASH

D.1 Introduction

This annex describes the delivery of DTS-UHD audio using the ISO/IEC 23009-1 [5] (MPEG-DASH) specification. It includes an overview of the general considerations for preparing streams for DASH delivery and a summary of DASH MPD parameter settings for DTS-UHD.

D.2 General considerations

Currently there are two defined classes of DTS-UHD client which correspond to the profiles defined in Annex F. Client devices advertising Profile 2 compatibility cannot play Profile 3 streams. Streams advertising *DecoderProfile* = 3 cannot be played on Profile 2 clients. Profile 3 clients can play both Profile 2 and Profile 3 streams.

Audio playback can only begin on a sync frame. DTS-UHD DASH sessions should always start on a sync frame. DTS-UHD clients will produce a decoded audio output on the first sync frame.

If multiple DTS-UHD streams are grouped in an adaptation set, seamless switching can be achieved if the streams in the adaptation set have certain properties in common:

- The base sampling frequency shall not change
- The Audio Chunk Type shall not change
- The Audio frame duration shall not change
- *RepresentationType* shall not change
- *NumPresentationsCode* shall not change
- DTS-UHD tracks within a DASH Adaptation Set should populate the parameters of the `AudioSampleEntry` with identical parameters, i.e. normally those of the highest bitrate stream in the adaptation set. Specifically:
  - DecoderProfileCode
  - MaxPayloadCode

D.3 DASH MPD Parameters for DTS-UHD

Single stream delivery to DASH clients shall be according to ISO/IEC 23009-1 [5]. Table D-1 summarizes the MPD elements and attributes with consideration for DTS-UHD.

The role attribute designated as **`urn:mpeg:dash:role:2011`** and defined in ISO/IEC 23009-1 [5] shall be used to define role parameters.

Table D-1: DTS-UHD DASH Elements and Attributes

Element or Attribute Name	Description
@codecs	The value of the codecs attribute shall be created according to the syntax described in IETF RFC 6381 [9]. For DTS-UHD, @codecs is the associated 4cc with no additional suffix. if DecoderProfile = 2 then @codecs = 'dtsx' if DecoderProfile = 3 then @codecs = 'dtsy' For streams formatted according to ISO/IEC 14496-12 [4], @codecs can be read directly from the codingname of the sample entry. For streams formatted according to ISO/IEC 13818-1 [8], DecoderProfile can be read from the DTS-UHD descriptor.
AudioChannelConfiguration	DTS-UHD shall use the following scheme URI: tag:dts.com, 2018:uhd:audio_channel_configuration and the value shall be a string representing the ChannelMask as defined in Table B-5, presented in hexadecimal form.
RandomAccess	The type to be used with DTS-UHD shall be "closed", i.e. the SAP type is 1.
Language	The language indicated should equal the language of the dialogue element.
Role	The Role@value should be set by the content author.
Label	The Label for a Representation should be set by the content author.

D.4 Support for Preselections

DASH delivery of multi-program and multi-stream presentations require the use of the Preselection Descriptor, defined by the schema **`urn:mpeg:dash:preselection:2016`** as defined in DASH [5].

The *tag* attribute used to describe the Preselection Element shall signal the DTS-UHD Presentation or a list of Object IDs for each of the delivered streams in the following manner:

- If the parameter is intended to be interpreted as a Presentation ID (*ucDesiredAuPresIndex)*, then the Preselection *tag* value shall be a string containing a single parameter comprising the letter 'p' (or 'P') appended with a numerical value equal to the intended Presentation ID value with no white space between the letter and the number. The DTS-UHD system shall use the `API_PRES_SELECT_SPECIFIC_AP` player API call to the decoder using the provided numerical portion of the parameter.

- If the parameter is intended to be interpreted as a list of one or more Object IDs (*unDesiredObjIDList*), then the Preselection *tag* value shall be a string of one more numerical parameter values separated by white space as described in DASH [5]. The DTS-UHD system shall interpret each parameter as an Object Id as associated with the `API_PRES_SELECT_OBJECT_ID_LIST` player API call to the decoder.

All other rules and conventions associated with Preselections are defined in DASH [5].

D.5 Example MPD Using Preselections

Table D-2 shows a an example of a single DTS-UHD Profile 3 stream carrying 2 Preselections and being selected using the Presentation ID.

Table D-2: MPD Example selecting by presentation ID

```
  <Period>
    <AdaptationSet id="1" mimeType="audio/mp4" codecs="dtsy" audioSamplingRate="48000" lang="en"
segmentAlignment="true" startWithSAP="1">
      <SupplementalProperty schemeIdUri="urn:mpeg:dash:preselection:2016" />
      <Role schemeIdUri="urn:mpeg:dash:role:2011" value="main"/>
      <SegmentTemplate timescale="48000" media="one_stream/$Time$.m4s"
initialization="one_stream/init.mp4">
        <SegmentTimeline>
          <S t="0" d="96000" r="36"/>
        </SegmentTimeline>
      </SegmentTemplate>
      <Representation id="r0" bandwidth="256000">
        <AudioChannelConfiguration schemeIdUri="tag:dts.com,2018:uhd:audio_channel_configuration"
value="180a03f"/>
      </Representation>
    </AdaptationSet>
    <!-- Preselection Element - Primary Preselection 48kHz, DTS-UHD, 5.1.4ch -->
    <Preselection id="10" tag="P0" preselectionComponents="1" codecs="dtsy"
audioSamplingRate="48000" lang="en">
      <Role schemeIdUri="urn:mpeg:dash:role:2011" value="main"/>
      <AudioChannelConfiguration schemeIdUri="tag:dts.com,2018:uhd:audio_channel_configuration"
value="180a03f"/>
    </Preselection>

    <!-- Preselection Element - French Language Preselection 48kHz, DTS-UHD 5.1.4ch -->
    <Preselection id="20" tag="P1" preselectionComponents="1" codecs="dtsy"
audioSamplingRate="48000" lang="fr">
      <Role schemeIdUri="urn:mpeg:dash:role:2011" value="dub"/>
      <AudioChannelConfiguration schemeIdUri="tag:dts.com,2018:uhd:audio_channel_configuration"
value="180a03f"/>
    </Preselection>
</Period>
```

Annex E (normative): DTS-UHD CMAF Profiles

E.1 Overview

This annex defines DTS-UHD Media Profiles that are compliant with ISO/IEC 23000-19 [6] (MPEG CMAF).

E.2 General Guidelines for DTS-UHD CMAF Tracks

E.2.1 CMAF Compliance

DTS-UHD tracks advertising CMAF compliance shall conform to Annex E.

E.2.2 Codec Profiles and Levels

DTS-UHD Profile 2 and DTS-UHD Profile 3 are defined in Annex F.

E.2.3 Media Access Unit Mapping to Media Samples

The SampleEntry for DTS-UHD CMAF tracks shall conform to Annex D. One media sample corresponds to one DTS-UHD audio frame.

E.2.4 Media Access Unit Sequence Mapping to CMAF Fragments

A DTS-UHD media sample is may be either a DTS-UHD synch frame or non-sync frame. The first DTS-UHD sample in a CMAF fragment shall be a sync-frame.

E.2.5 CMAF Track Constraints for CMAF Switching Sets

DTS-UHD tracks shall be constrained according to clause D.2 when seamless switching is required. The restrictions apply to all audio samples within a track, and to tracks within a switching set where seamless switching is required.

E.2.6 CMAF Media Profile Internet Media Type

DTS-UHD CMAF track files shall be signaled with the Internet Media Type as indicated in Table D-1.

E.2.7 CMAF Media Profile Brand

The file type box ('`ftyp`') of a DTS-UHD track may contain the brand '`dts2`', '`dts3`' or '`dts4`'.

If the '`dts2`' brand is present in the file type box, then the track shall comply with the requirements of the definition described in the present annex and DTS-UHD Profile 2 as described in Annex F.

If the '`dts3`' brand is present in the file type box, then the track shall comply with the requirements of the definition described in the present annex and DTS-UHD Profile 3 as described in Annex F with the additional constraints that:

- only Audio Chunk Types = 0x01, 0x03 or 0x04 shall be available;
- a default preselecton shall be playable from a single bitstream (i.e. if multi-stream playback is available, it shall be optional).

If the 'dts4' brand is present in the file type box, then the constraints of the 'dts3' brand shall apply, with the exception that multi-stream support is required to render the default audio preselection.

E.3 Guidelines for DTS-UHD CMAF Media Profiles

E.3.1 General

In testing CMAF players, if the player under test is intended to comply with the 'dts2' brand, then a single DTS-UHD Profile 2 compliant stream is required.

If the player under test is intended to comply with the 'dts3' or 'dts4' brand, then a DTS-UHD Profile 3 stream, as defined in clause E, is required.

E.3.2 Audio Track Format

A DTS-UHD track conforming to the Media Profile defined by the 'dts2' brand shall conform to clause 10.2 of ISO/IEC 23000-19 [6], Annex A of the present document and DTS-UHD Profile 2 as defined in clause F.2.

A DTS-UHD track conforming to the Media Profile defined by the 'dts3' brand shall conform to clause 10.2 of ISO/IEC 23000-19 [6], Annex A of the present document and DTS-UHD Profile 3 as defined in clause F.3.

E.3.3 Loudness and Dynamic Range Control

The DTS-UHD bitstream is capable of carrying multiple loudness parameters, some of which include (nominally) the complete preselection, just the speech objects, and composition of all components excluding the speech objects. The content encoder may use these to signal loudness values and the method(s) used to calculate the values.

Multiple measurement methods are supported, and the bitstream syntax has provisions to carry as many as 16 sets of loudness parameters.

Each set of loudness parameters consists of 3 values:

m_rLoudness is the measured loudness, stored with an associated asset type code

m_ucAssociatedAssetType which describes the type of asset being measured

m_ucLoudnessMsrmType which indicates the method used to calculate m_rLoudness.

Details regarding loudness measurements and signaling for DTS-UHD are documented in clause 7.7.6.5.

E.3.4 Audio Parameters

All parameters are needed to initialize the DTS-UHD decoder are provided in a sync sample.

E.3.5 Audio Presentation Time Adjustment

Delay can be compensated prior to stream encapsulation using the techniques specified in ISO/IEC 23000-19 [6], clause G.5. The use of EditListBox is not required and further delay compensation in the receiver is not required.

E.4 Delivery Considerations for DTS-UHD CMAF Tracks

The considerations for seamless switching and smooth switching for DASH, as described in Annex C, apply to the delivery of CMAF tracks.

E.5 Playback Considerations for DTS-UHD CMAF Tracks

When playing back DTS-UHD tracks in a supported MSE (HTML 5 Web Browser) environment, no special considerations are necessary for playback or segment boundaries.

Annex F (normative): DTS-UHD Media Profiles

F.1 Overview of Profiles

DTS-UHD bitstreams can carry many waveforms that can be organized into groups and presentations, therefore it is necessary to consider profiles to accommodate the physical capabilities of common commercial platforms. The constraints of a given system are mainly determined by the processing capability of the CPU engine and the amount of available memory. To enable the interoperability between devices (players) and content (bitstreams), some specific profiles have been defined that consider the features needed and the platform capability of commercial products.

Specific application profiles, such as DVB broadcast, may apply additional constraints to the bitstreams due to other system considerations.

F.2 DTS-UHD Profile 2 Bitstreams

F.2.1 Profile 2 Overview

Profile 2 bitstreams are mainly characterized as a bitstream containing a single presentation with a channel configuration of up to 5.1.4 channel-based waveforms.

F.2.2 Profile 2 Bitstream Requirements

Profile 2 bitstreams shall be limited to 9 full bandwidth channels and up to 1 LFE channel.

Profile 2 bitstreams shall have the bitstream parameter *bFullChannelBasedMixFlag* is set to 1.

A stream with *bFullChannelBasedMixFlag* == 1 has several implications on the bitstream constraints, (as defined in clauses 6 and 7), including the following:

- The bitstream will contain a single presentation whose waveform locations are described by a channel mask.
- Only audio chunks of type 0x01 will be present.
- No waveforms with locations defined using spatial coordinates or soundfields will be present.

F.2.3 Profile 2 AudioSampleEntry

Requirements imposed on `DTSUHDSampleEntry` for bitstreams compliant with Profile 2 are as follows:

- *codingname* shall be set to '`dtsx`'
- *DecoderProfileCode* shall be set to 0

The result of Profile 2 requirements has the following impact on the `DTSUHDSampleEntry` as defined in Annex A:

- FrameDurationCode = 1
- MaxPayloadCode = 0
- NumPresentationsCode = 1
- BaseSamplingFreqCode = 1
- SampleRateMod = 0

- StreamIndex = 0
- IDTagPresent = 0

F.3 DTS-UHD Profile 3 Bitstreams

F.3.1 Profile 3 Overview

Profile 3 is unconstrained other than the total number of waveforms that are required to be decoded in a single presentation. Profile 3 decoder systems include an audio renderer, so the use of audio objects and audio object metadata are supported.

F.3.2 Profile 3 Bitstream Requirements

Profile 3 bitstreams shall conform to the present document. Content authors should consider the minimum requirements for Profile 3 devices as defined in clause F.4.3, particularly that, of the active objects, only the first 16 waveforms can be guaranteed to play back on a Profile 3 capable player.

F.3.3 Profile 3 AudioSampleEntry

Requirements imposed on the `AudioSampleEntry` for bitstreams compliant with Profile 3 are as follows:

- *codingname* shall be set to '`dtsy`'
- *DecoderProfileCode* shall be set to 1

F.4 DTS-UHD Device Requirements

F.4.1 About DTS-UHD Devices

To meet a minimum level of interopeability, it is necessary to define a minimum expected capability of a device advertising the ability to play DTS-UHD streams.

Specific ecosystems may further define how specific features are implemented in devices.

F.4.2 DTS-UHD Profile 2 Device Requirements

DTS-UHD Profile 2 Devices shall meet the following minimum requirements:

- Shall be able to decode a stream compliant with DTS-UHD Profile 2 of up to 9 full bandwith waveforms that are played to channels, plus 1 LFE channel.
- Shall be capable of downmixing the decoded waveforms to to the number of speaker feeds supported by the device, and any sub-combination of speaker feeds supported by the device (including stereo).

F.4.3 DTS-UHD Profile 3 Device Requirements

DTS-UHD Profile 3 Devices shall meet the requirements in clause F.4.2 and additionally:

- Shall be capable of selecting a presentation with a single DTS-UHD Profile 3 stream and decoding components totalling a minimum of 16 full bandwidth waveforms that may be described as channels, 3D sound objects or soundfields, plus 2 LFE channels.

- Shall be able to render all decoded waveforms using the either the metadata provided in the stream, or, if the device supports object interactivity, metadata that has been over riddden by the player system.
- Shall be capable of downmixing the decoded waveforms to a stereo output.
- Shall be able to downmix the decoded waveforms to the number of speaker feeds supported by the device, or any sub-combination of speakerfeeds supported by the device.
- Support for decoding more than 16 waveforms is optional. Audio programs designed with more than 16 waveforms should be authored in consideration of that constraint. The decoder shall sequentially process the components in the order of the list of *ObjectId*s, whether that list is associated with the *PresentationId* (as described in clause 6.4.10), or in explicit list of *ObjectIds* associated with the API call `API_PRES_SELECT_OBJECT_ID_LIST`. If the decoder is unable to decode the waveforms associated with next *ObjectId* listed, the decoder shall stop processing and present the waveforms it has decoded to the renderer.

F.4.4 DTS-UHD Profile 3 Multi-stream Device Requirements

Support for audio delivered in more than one audio stream is primarily dependent on the playback environment. Currently, some players that are dependent on MSE or EME APIs are not able to associate multiple streams with an audio program, so DTS-UHD streams intended for this environment should carry a suitable default presentation in the primary audio stream. A Profile 3 decoder can decode at least 16 waveforms without regard to how they were delivered.

DTS-UHD Profile 3 Devices supporting multi-stream shall meet the requirements in clause F.4.3 and additionally:

- Shall be capable of processing 16 full bandwidth waveforms and 2 LFE channels from up to 16 streams.

Annex G (informative): Multi-stream Considerations

G.1 Introduction

This clause describes the design considerations for a player supporting DTS-UHD content delivered with multiple streams. The applications for multi-stream can be as straightforward as a commentary track layered onto to the main audio track to language substitution and even interactivity. The basic architecture for a DTS-UHD multi-stream player is described in this annex. Systems designed to accommodate multi-stream audio may have additional constrains in consideration of the intended use cases, such as DVB broadcast.

G.2 Multi-stream Architecture

DTS-UHD streams that are constructed for multi-stream delivery may have an associated stream identifier. This might be necessary to establish a hierarchy for the metadata selection by the renderer.

Figure G-1 illustrates a simple multi-stream session with 3 streams.

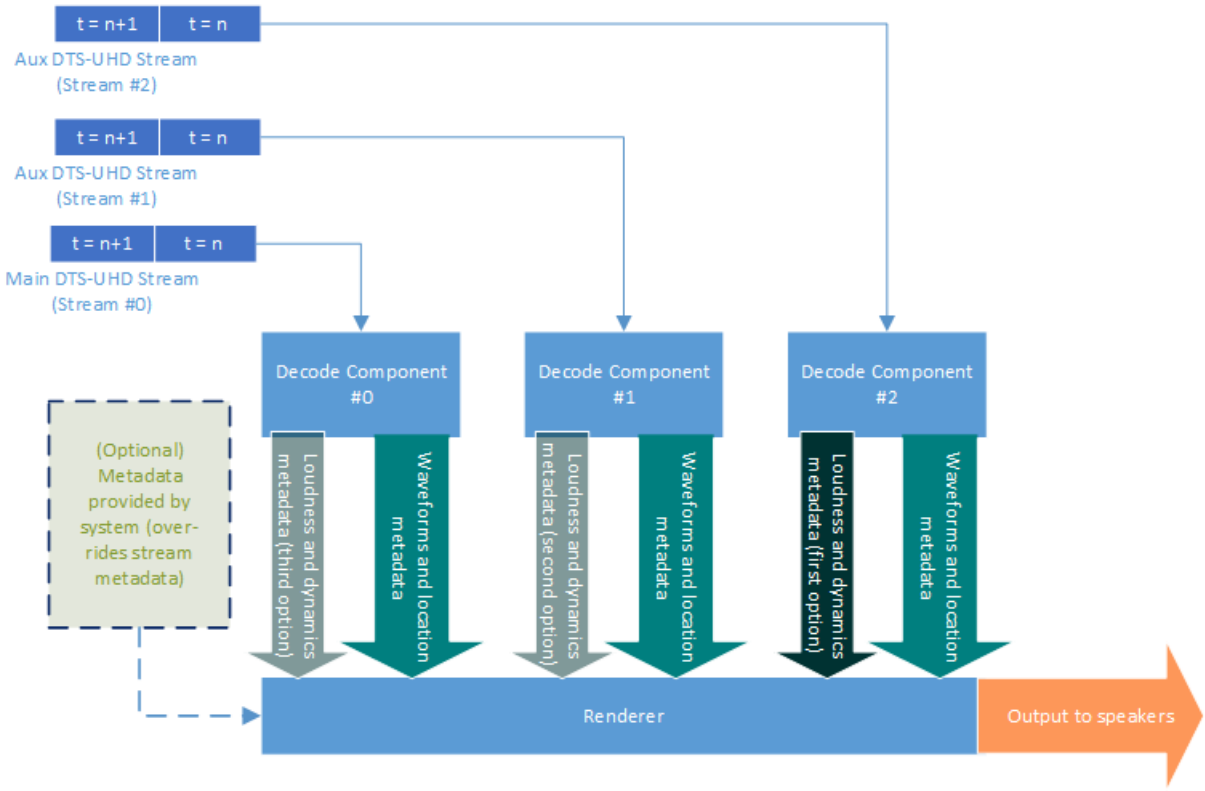

Figure G-1: DTS-UHD Multi-stream Example

In a multi-stream device, the audio frames for a given presentation time are delivered sequentially to the decoder system, with the audio frame corresponding to Stream #0 (in the above example), followed by the frame from Stream #1 and Stream #2 in order. The decoder itself does not see the StreamID, it only considers the order in which the audio frames for a given presentation time are delivered. So other methods of controlling this ordering, such as using the DASH MPD, can be applied. This will set the component priority and the metadata priority.

Each frame from a given stream may contain multiple presentations and components, so the identification of a single component or set of components may require additional parameters, such as a presentation identifier or a list of one or more object identifiers. Each decoder session will decode the associated waveforms from each stream and pass the result, along with the associated metadata, to the renderer. A method of managing these parameters using the tools provided in DASH are described in clause D.4.

The component priority is such that the first component delivered has the highest decoding priority, and subsequent components follow in priority respectively. This only comes into play in use cases when a given presentation contains more than the minimum number of waveforms that a system is required to handle (i.e more than 16 waveforms). The metadata for components not processed by the decoder are not forwarded to the renderer.

The renderer will apply the object metadata when all waveforms and the accompanying metadata for a give presentation time have been received. The component metadata, for example, output scaling, will override such metadata from earlier components. If the system overrides some metadata parameters, such as the assigned location or relative loudness, the renderer will ignore subsequent updates from the stream and apply the overridden parameters until otherwise instructed.

Annex H (informative): Registrations for DTS-UHD

H.1 Overview

This clause lists some select registrations that may be of interest to users of the DTS-UHD codec.

H.2 MP4RA

Registrations related to DTS-UHD that are listed in the MPEG-4 Registration Authority, which can be found at http://mp4ra.org, are listed below.

The following are listed as codecs with an ObjectType assignment:

dtsx

dtsy

The following are listed as brands:

dts2

dts3

H.3 IANA

IANA registrations related to DTS-UHD are listed below.

The definition of vnd.dts.uhd can be found at https://www.iana.org/assignments/media-types/audio/vnd.dts.uhd.

History

Document history		
V1.1.1	April 2017	Publication
V1.2.1	May 2019	Publication