INTERNATIONAL STANDARD

ISO/IEC 23003-4

First edition
2015-11-15

AMENDMENT 1
2017-12

Information technology — MPEG audio technologies —

Part 4: Dynamic Range Control

AMENDMENT 1: Parametric DRC, gain mapping and equalization tools

Technologies de l'information — Technologies audio MPEG —

Partie 4: Contrôle de gamme dynamique

AMENDEMENT 1: Outils de DRC paramétrique, de mappage des gains et d'égalisation

Reference number
ISO/IEC 23003-4:2015/Amd.1:2017(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, SC 29, *Coding of audio, picture, multimedia and hypermedia information.*

A list of all parts in the ISO/IEC 23003 series can be found on the ISO website.

Information technology — MPEG audio technologies —

Part 4:
Dynamic Range Control

AMENDMENT 1: Parametric DRC, gain mapping and equalization tools

Page vi, Introduction

Add the following at the end of the Introduction:

Loudness normalization is fully integrated with DRC and peak control to avoid clipping. A metadata-controlled equalization tool is provided to compensate for playback scenarios that impact the spectral balance, such as downmix or DRC. Furthermore, the DRC tool supports metadata-based loudness equalization to compensate the effect of playback level changes on the spectral balance.

Page 2, Clause 4

Insert the following new definitions and maintain the alphabetical order:

mod modulo operator: (x mod y) = x-y floor (x/y)

sizeof(x) size operator that returns the bit size of a field

Page 3, Clause 5

Replace the first paragraph:

The technology described in this part of ISO/IEC 23003 is called DRC tool. It provides efficient control of dynamic range, loudness, and clipping based on metadata generated at the encoder. The decoder can choose to selectively apply the metadata to the audio signal to achieve a desired result. Metadata for dynamic range compression consists of encoded time-varying gain values that can be applied to the audio signal. Hence, the main blocks of the DRC tool include a DRC gain encoder, a DRC gain decoder, a DRC gain modification block, and a DRC gain application block. These blocks are exercised on a frame-by-frame basis during audio processing. Various DRC configurations can be conveyed in a separate bitstream element, such as configurations for a downmix or combined DRCs. The DRC set selection block decides based on the playback scenario and the applicable DRC configurations which DRC gains to apply to the audio signal. Moreover, the DRC tool supports loudness normalization based on loudness metadata.

With:

The technology described in this document is called the “DRC tool”. It provides efficient control of dynamic range, loudness, and clipping based on metadata generated at the encoder. The decoder can choose to selectively apply the metadata to the audio signal to achieve a desired result. Metadata for dynamic range compression consists of encoded time-varying gain values that can be applied to the audio signal. Hence, the main blocks of the DRC tool include a DRC gain encoder, a DRC gain decoder, a DRC gain modification block, and a DRC gain application block. These blocks are exercised on a frame-by-frame basis during audio processing. In addition to encoded time-varying gain values, the DRC

gain decoder can also receive parametric DRC metadata for generation of time-varying gain values at the decoder. Various DRC configurations can be conveyed in a separate bitstream element, such as configurations for a downmix or combined DRCs. The DRC set selection block decides based on the playback scenario and the applicable DRC configurations which DRC gains to apply to the audio signal. Moreover, the DRC tool supports loudness normalization based on loudness metadata.

Page 3, Clause 5

Add the following at the end of the clause:

The DRC tool provides support for loudness equalization, or sometimes called “loudness compensation”, that can be applied to compensate for the effect of the playback level on the spectral balance. For this purpose, time-varying loudness information can be recovered from DRC gain sequences to dynamically control the compensation module. While the compensation module is out of scope, the interface describes in which frequency ranges the loudness information should be applied.

A flexible tool for generic metadata-controlled equalization is provided. The tool can be used to reach the desired spectral balance of the reproduced audio signal depending on a wide variety of playback scenarios, such as downmix, DRC, or playback room size. It can operate in the sub-band domain of an audio decoder and in the time domain.

Page 4, 6.1.1

Replace the following list after Table 1:

The static payload is divided into five logical blocks:

— channelLayout()

— downmixInstructions ()

— drcCoefficientsBasic(), drcCoefficientsUniDrc()

— drcInstructionsBasic(), drcInstructionUniDrc()

— loudnessInfo()

With:

The static payload is divided into six logical blocks:

— channelLayout();

— downmixInstructions(), downmixInstructionsV1();

— drcCoefficientsBasic(), drcCoefficientsUniDrc(), drcCoefficientsUniDrcV1();

— drcInstructionsBasic(), drcInstructionUniDrc(), drcInstructionUniDrcV1();

— loudnessInfo(), loudnessInfoV1();

— loudEqInstructions().

Page 4, 6.1.1

Replace the last two paragraphs:

uniDrcConfig() contains all blocks except for the loudnessInfo() blocks which are bundled in loudnessInfoSet(). The last part of the uniDrcConfig() payload can include future extension payloads. In the event that a *uniDrcConfigExtType* value is received that is not equal to UNIDRCCONFEXT_TERM, the DRC tool parser shall read and discard the bits (otherBit) of the extension payload. Similarly, the last part of the loudnessInfoSet() payload can include future extension payloads. In the event that a *loudnessInfoSetExtType* value is received that is not equal to UNIDRCLOUDEXT_TERM, the DRC tool parser shall read and discard the bits (otherBit) of the extension payload.

The top level fields of uniDrcConfig() include the audio sample rate, which is a fundamental parameter for the decoding process (if not present, the audio sample rate is inherited from the employed audio codec). Moreover, the top level fields of uniDrcConfig() include the number of instances of each of the logical blocks, except for the channelLayout() block which appears only once. The top level fields of loudnessInfoSet() only include the number of loudnessInfo() blocks. The five logical blocks are described in the following.

With:

uniDrcConfig() contains all blocks except for the loudnessInfo() blocks which are bundled in loudnessInfoSet(). The last part of the uniDrcConfig() payload can include future extension payloads. In the event that a *uniDrcConfigExtType* value is received that is not equal to UNIDRCCONFEXT_TERM, the DRC tool parser shall read and discard the bits (otherBit) of the extension payload. Similarly, the last part of the loudnessInfoSet() payload can include future extension payloads. In the event that a *loudnessInfoSetExtType* value is received that is not equal to UNIDRCLOUDEXT_TERM, the DRC tool parser shall read and discard the bits (otherBit) of the extension payload. Each extension payload type in uniDrcConfig() or loudnessInfoSet() shall not appear more than once in the bitstream if not stated otherwise. An extension payload of type UNIDRCCONFEXT_V1 shall preceed an extension payload of type UNIDRCCONFEXT_PARAM_DRC in the bitstream if both payloads are present. Note that for ISO/IEC 14496-12, configuration extension payloads are provided according to Table AMD1.26.

The top level fields of uniDrcConfig() include the audio sample rate, which is a fundamental parameter for the decoding process (if not present, the audio sample rate is inherited from the employed audio codec). Moreover, the top level fields of uniDrcConfig() include the number of instances of each of the logical blocks, except for the channelLayout() block which appears only once. The top level fields of loudnessInfoSet() only include the number of loudnessInfo() blocks. The six logical blocks are described in the following.

Page 5, 6.1.2.2

Replace:

6.1.2.2 downmixInstructions()

This block includes a unique non-zero downmix identifier (downmixId) that can be used externally to refer to this downmix. The targetChannelCount specifies the number of channels after downmixing to the target layout. It may also contain downmix coefficients, unless they are specified elsewhere. For use cases where the base audio signal represents objects or other audio content, the downmixId can be used to refer to a specific target channel configuration of a present rendering engine.

With:

6.1.2.2 downmixInstructions() and downmixInstructionsV1()

This block includes a unique non-zero downmix identifier (*downmixId*) that can be used externally to refer to this downmix. The *targetChannelCount* specifies the number of channels after downmixing to the target layout. It may also contain downmix coefficients, unless they are specified elsewhere. For

use cases where the base audio signal represents objects or other audio content, the *downmixId* can be used to refer to a specific target channel configuration of a present rendering engine. In contrast to downmixInstructions(), the downmixInstructionsV1() payload includes an offset for all downmix coefficients and the coefficient decoding does not depend on the LFE channel assignment. The downmixInstructions() box for ISO/IEC 14496-12 contains the corresponding metadata of either one of the in-stream payloads as indicated by the *version* parameter of the box.

Page 5, 6.1.2.3

Replace the heading of 6.1.2.3 with:

6.1.2.3 drcCoefficientsBasic(), drcCoefficientsUniDrc(), and drcCoefficientsUniDrcV1()

Page 5, 6.1.2.3

Replace the second paragraph with:

The DRC location field encoding depends on the audio codec. A codec specification may include this specification, and use values 1 to 4 to refer to codec-specific locations as indicated in Table 2. For example, for AAC (ISO/IEC 14496-3), the codec-specific values of the DRC location field are encoded as shown in Table 3.

Page 6, 6.1.2.3

Add new paragraph before 6.1.2.4:

The drcCoefficientsUniDrcV1() payload is defined in Table AMD1.24. It contains the same information as drcCoefficientsUniDrc() except for the assignment of DRC gain sequences to gain sets and the optional specification of a number of parametric DRC characteristics. The drcCoefficientsUniDrc() payload assigns gain sequences in order of transmission. In contrast, the drcCoefficientsUniDrcV1() payload maps a gain sequence by index to *gainSets*. The latter permits to refer to the same gain sequence for multiple DRC bands which is not possible when using drcCoefficientsUniDrc(). If a drcCoefficientsUniDrcV1() payload is present, any drcCoefficientsUniDrc() payload for the same location is ignored.

The drcCoefficientsUniDrcV1() payload can also include information about dynamic equalization filters if the field *shapingFiltersPresent*==1. There can be a number of filters that are indexed in order of appearance. The DRC sets defined in drcInstructionsUniDrcV1() can refer to specific filters using their indices (see 6.4.11).

Page 6, 6.1.2.4

Replace the heading of 6.1.2.4 with:

6.1.2.4 drcInstructionsBasic(), drcInstructionsUniDrc(), and drcInstructionsUniDrcV1()

Page 6, 6.1.2.4

Insert the following after the first paragraph of 6.1.2.4:

The drcInstructionsUniDrcV1() payload is defined in Table AMD1.25. Compared to the drcInstructionsUniDrc() payload, it contains the same information plus several enhancements. However, the *downmixIDs* that appear in the two payloads are interpreted differently. For

drcInstructionsUniDrcV1(), the *downmixIDs* indicate which downmix configuration is permitted in combination with this DRC, but it does not specify whether the DRC is applied to the downmix or the base layout. This is controlled by the *drcApplyToDownmix* flag instead.

The enhanced metadata of drcInstructionsUniDrcV1() compared to drcInstructionsUniDrc() includes references to target DRC characteristics and dynamic equalization filters. If a target characteristic is referenced, the corresponding DRC gain values shall be mapped to the target characteristic unless the host system overrides the target characteristic as specified in Table 16. If dynamic equalization filters are referenced, they shall be applied when the corresponding DRC set is selected and when this feature is supported by the decoder implementation (see 6.4.11).

Page 7, 6.1.2.4

Replace the paragraphs:

A second DRC set may be specified for certain configurations. These configurations include cases where, e.g. one DRC set is used for dynamic range compression and the other for clipping prevention ("Clipping" bit is set); or, e.g. one DRC set is applied before and the other after the downmix. In those cases, the second DRC set contains a non-zero field *dependsOnDrcSet* that has the value of the *drcSetId* of the first DRC set it depends on. The declared DRC set effects of the second DRC set do not take into account the effects of the first DRC set. If the first DRC set is not designed to be used without combining it with another DRC set, the *noIndependentUse* flag shall be set to 1. In that case, the DRC set can only be used in combination with another DRC set as indicated by the *dependsOnDrcSet* field of the other set that is combined with it.

Usually, each audio channel is assigned to a DRC gain sequence. A collection of channels assigned to the same DRC gain sequence is called "channel group". The assignment of a DRC gain sequence to a channel group is done in the order of first appearance of the sequence index when iterating through all channels (see also Table 14). A DRC gain sequence index *bsSequenceIndex*==0 indicates that the assigned channel will be passed through by the DRC tool without processing unless otherwise noted. Note that therefore *bsSequenceIndex* is effectively 1-based, whereas the corresponding indices (*sequenceIndex*) for processing are zero-based.

If subsequent channels are assigned the same sequence index, the field *repeatSequenceCount* indicates how many channels will have the same sequence not including the first.

With:

A second DRC set may be specified for certain configurations. These configurations include cases where, for example, one DRC set is used for dynamic range compression and the other for clipping prevention ("Clipping" bit is set); or, for example, one DRC set is applied before and the other after the downmix. In those cases, the first DRC set contains a non-zero field *dependsOnDrcSet* that has the value of the *drcSetId* of the second DRC set it depends on. The declared DRC set effects of the first DRC set do not take into account the effects of the second DRC set. If the second DRC set is not designed to be used without combining it with another DRC set, the *noIndependentUse* flag shall be set to 1. In that case, the DRC set can only be used in combination with another DRC set as indicated by the *dependsOnDrcSet* field of the other set that is combined with it.

If a second DRC is specified by the *dependsOnDrcSet* field of a drcInstructionsUniDrcV1() payload, the combined DRCs cannot be applied if the DRC decoder downmix configuration does not match any *downmixID* in any of the two corresponding drcInstructionsUniDrcV1() payloads. Please note that these downmix configurations may include the output of the base layout without downmix, which is specified by a *downmixID*==0. Otherwise, the combined DRC is compatible with the downmix.

A DRC gain set is a set of DRC gain sequences, where the sequences are assigned to the bands of a multi-band DRC. For a single band DRC, the gain set contains just one gain sequence. Usually, each audio channel is assigned to a DRC gain set. A collection of channels assigned to the same DRC gain set is called "channel group". The assignment of a DRC gain set to a channel group is done in the order of first appearance of the set index when iterating through all channels (see also Table 14). A DRC gain set index

bsGainSetIndex==0 indicates that the assigned channel will be passed through by the DRC tool without processing unless otherwise noted. Note that therefore, *bsGainSetIndex* is effectively 1-based, whereas the corresponding indices (*gainSetIndex*) are zero-based.

If subsequent channels are assigned the same gain set index, the field *repeatGainSetCount* indicates how many channels will have the same gain set not including the first.

Page 7, 6.1.2.5

Replace the heading of 6.1.2.5 with:

6.1.2.5 loudnessInfo() and loudnessInfoV1()

Page 8, 6.1.2.5

Replace the second paragraph:

If *downmixId* is zero, then loudnessInfo() applies to the base layout. If the *drcSetId* is zero, then loudnessInfo() applies to the audio signal without DRC processing.

With:

If *downmixId* is zero, then loudnessInfo() applies to the base layout. If the *drcSetId* is zero, then loudnessInfo() applies to the audio signal without DRC processing. If *drcSetId* is 0x3F, then loudnessInfo() applies to any DRC processing including no DRC.

Page 8, 6.1.2.5

Replace the third paragraph:

The fields *samplePeakLevel* and *truePeakLevel* represent the level of the maximum sample magnitude in dBFS and the true peak in dBTP, respectively, of the associated audio content before or after audio encoding as defined in Reference [4]. The *measurementSystem* field includes standardized systems and others (see Table A.37). System 3 is defined as ITU-R BS.1770-3 with pre-processing. The pre-processing is a high-pass filter that models the typical limited frequency response of portable device loudspeakers. System 4 is defined as "User". It means that the corresponding *methodValue* reflects a (subjective) user preference. System 5 is defined as "Expert/Panel". It means that the corresponding *methodValue* represents a (subjective) expert or panel preference.

With:

The fields *samplePeakLevel* and *truePeakLevel* represent the level of the maximum sample magnitude in dBFS and the true peak in dBTP, respectively, of the associated audio content before or after audio encoding as defined in Reference [4]. The *measurementSystem* field includes standardized systems and others (see Table A.37). System 3 is defined as ITU-R BS.1770-4 with pre-processing. The pre-processing is a high-pass filter that models the typical limited frequency response of portable device loudspeakers. System 4 is defined as "User". It means that the corresponding *methodValue* reflects a (subjective) user preference. System 5 is defined as "Expert/Panel". It means that the corresponding *methodValue* represents a (subjective) expert or panel preference.

Page 8, 6.1.2.5

Add a new subclause after 6.1.2.5:

6.1.2.6 loudEqInstructions()

The loudEqInstructions() payload includes information relevant for the loudness equalization support. Each instance of this payload defines a set of loudness equalizer metadata and which combinations of downmix, DRC, and EQ it can be applied to. The metadata includes references to the corresponding DRC gain sequences and associated parameters needed to derive the acoustic level data. A typical way of generation and use of the metadata is given in D.2.10.

Page 11, 6.1.3

Replace the paragraph before Table 7:

Table 6 includes functions to check the availability and to retrieve peak-related information from loudnessInfo() and a drcInstructions block which can have the basic or uniDrc format. Table 7 shows pseudo code for some of the functions for the truePeakLevel and limiterPeakTarget. The functions for samplePeakLevel can be implemented by replacing truePeakLevel with samplePeakLevel.

With:

Table 6 includes functions to check the availability and to retrieve peak-related information from loudnessInfo() and a drcInstructions block which can have the basic or uniDrc format. Table 7 shows pseudo code for some of the functions for the *truePeakLevel* and *limiterPeakTarget* based on downmixInstructions(), drcInstructionsUniDrc(), and loudnessInfo() payloads. The functions shall be adapted accordingly for the downmixInstructionsV1(), drcInstructionsUniDrcV1(), or loudnessInfoV1() payloads, if present. The functions for *samplePeakLevel* can be implemented by replacing *truePeakLevel* with *samplePeakLevel*.

Page 9, 6.1.3

Add the following paragraph and table after Table 5:

The applicable loudnessInfo() structure for determination of *programLoudness* and *anchorLoudness* is determined as specified in Table AMD1.1. The final loudness metadata value is extracted based on the selection steps specified in 6.6.2.

Table AMD1.1 — Determination of applicable loudnessInfo() structure for selection of programLoudness or anchorLoudness for a specific DRC set

```
getApplicableLoudnessInfoStructure(drcSetId, downmixIdRequested) {
  /* default value */
  loudnessInfo = getLoudnessInfoStructure(drcSetId, downmixIdRequested);
  /* fallback values */
  if (loudnessInfo == NULL) {
    loudnessInfo = getLoudnessInfoStructure(drcSetId, 0x7F);
  } else if (loudnessInfo == NULL) {
    loudnessInfo = getLoudnessInfoStructure(0x3F, downmixIdRequested);
  } else if (loudnessInfo == NULL) {
    loudnessInfo = getLoudnessInfoStructure(0, downmixIdRequested);
  } else if (loudnessInfo == NULL) {
    loudnessInfo = getLoudnessInfoStructure(0x3F, 0x7F);
  } else if (loudnessInfo == NULL) {
```

Table AMD1.1 *(continued)*

```
    loudnessInfo = getLoudnessInfoStructure(0, 0x7F);
  } else if (loudnessInfo == NULL) {
    loudnessInfo = getLoudnessInfoStructure(drcSetId, 0);
  } else if (loudnessInfo == NULL) {
    loudnessInfo = getLoudnessInfoStructure(0x3F, 0);
  } else if (loudnessInfo == NULL) {
    loudnessInfo = getLoudnessInfoStructure(0, 0);
  }
  return loudnessInfo;
}
getLoudnessInfoStructure(drcSetId, downmixId) {
  if (useAlbumMode == 1) count = loudnessInfoAlbumCount;
  else count = loudnessInfoCount;
  for (i=0; i<count; i++) {
    if (loudnessInfo[i]->drcSetId == drcSetId) &&
       (loudnessInfo[i]->downmixId == downmixId) {
      for (j=0; j<loudnessInfo[i]->measurmentCount; j++) {
        if ((loudnessInfo[i]->measure[j]->methodDefinition==1) ||
            (loudnessInfo[i]->measure[j]->methodDefinition==2)) {
          return loudnessInfo[i];
        }
      }
    }
  }
  return NULL;
}
```

Page 10, Table 6

Replace Table 6 with:

Table 6 — Determination of signalPeakLevel for a specific DRC set

```
getSignalPeakLevelForDrcSet (drcSetId, downmixIdRequested) {
  dmxId = downmixIdRequested;
  if truePeakLevelIsPresent(drcSetId, dmxId) {
    signalPeakLevel = getTruePeakLevel(drcSetId, dmxId);
  } else if samplePeakLevelIsPresent(drcSetId, dmxId) {
    signalPeakLevel = getSamplePeakLevel(drcSetId, dmxId);
  } else if truePeakLevelIsPresent(0x3F, dmxId) {
    signalPeakLevel = getTruePeakLevel(0x3F, dmxId);
  } else if samplePeakLevelIsPresent(0x3F, dmxId) {
    signalPeakLevel = getSamplePeakLevel(0x3F, dmxId);
  } else if limiterPeakTargetIsPresent(drcSetId, dmxId) {
    signalPeakLevel = getLimiterPeakTarget(drcSetId, dmxId);
  } else if (dmxId != 0) {
    signalPeakLevelTmp = 0.0;
    downmixPeakLevelLinear = 0.0;
    if downmixCoefficientsArePresent(dmxId) {
      for (i=0; i<targetChannelCount; i++) {
```

Table 6 *(continued)*

```
        downmixPeakLevelLinearTmp = 0.0;
        for (j=0; j<baseChannelCount; j++) {
          downmixPeakLevelLinearTmp +=
              pow(10.0, getDownmixCoefficient(dmxId, i, j)/20.0);
        }
        if (downmixPeakLevelLinear < downmixPeakLevelLinearTmp) {
          downmixPeakLevelLinear = downmixPeakLevelLinearTmp;
        }
      }
    }
    if truePeakLevelIsPresent(drcSetId, 0) {
      signalPeakLevelTmp = getTruePeakLevel(drcSetId, 0);
    } else if samplePeakLevelIsPresent(drcSetId, 0) {
      signalPeakLevelTmp = getSamplePeakLevel(drcSetId, 0);
    } else if truePeakLevelIsPresent(0x3F, 0) {
      signalPeakLevelTmp = getTruePeakLevel(0x3F, 0);
    } else if samplePeakLevelIsPresent(0x3F, 0) {
      signalPeakLevelTmp = getSamplePeakLevel(0x3F, 0);
    } else if limiterPeakTargetIsPresent(drcSetId, 0) {
      signalPeakLevelTmp = getLimiterPeakTarget(drcSetId, 0);
    }
    signalPeakLevel = signalPeakLevelTmp + 20.0*log10(downmixPeakLevelLinear);
  } else {
    signalPeakLevel = 0.0;      /* worst case estimate */
  }
  return signalPeakLevel
}
```

Page 12, 6.3.1

Replace the paragraph:

The most relevant metadata for the selection process is summarized in Table 8. The bit fields of the *drcSetEffect* field are described in detail in Table A.32. All parameters that can be supplied by the host to control loudness normalization and dynamic range compression are summarized in Table A.40 and Table A.41, respectively.

With:

The most relevant metadata for the selection process is summarized in Table 8. The bit fields of the *drcSetEffect* field are described in detail in Table A.32. All parameters that can be supplied by the host to control loudness normalization and dynamic range compression are summarized in Table A.40 and Table A.41, respectively.

If the DRC tool is configured to support equalization (EQ) according to 6.8, the selection process includes the EQ-related metadata and the final selection also considers requests for specific EQ, such as EQ dependent on the playback room size. Otherwise, the preselection discards all DRC sets that require EQ, i.e. which have a value of requiresEq==1.

Page 14, 6.3.2.1

Replace:

Table 9 — Requirements for DRC pre-selection

#	Requirement	Applicability	Comment
1	DownmixId of DRC set matches the requested downmixId.	If a downmixId is requested	See 6.3.2.2.
2	Output channel layout of DRC set matches the requested layout.	If a target channel layout is requested	See 6.3.2.2.
3	Channel count of DRC set matches the requested channel count.	If a target channel count is requested	See 6.3.2.2.
4	The DRC set is not a “Fade-” or “Ducking-” only DRC set.	Always	DRC sets with “Fade” or “Ducking” effect are selected automatically. They are not subject to this selection process.
5	The number of DRC bands is supported.	Always	DRC sets that exceed the number of supported DRC bands are discarded. For time-domain DRC, the maximum is four bands.
6	Independent use of DRC set is permitted.	If the DRC set is not used in combination with another DRC set.	DRC sets with a noIndependentUse flag value of 1 can only be used in combination with a second DRC set.
7	The range of the target loudness specified for a DRC set has to include the requested decoder target loudness.	If drcSetTargetLoudnessPresent==1 and no explicit peak information is available for that DRC set.	See 6.3.2.2.2.
8	Clipping is minimized.	Except for DRC sets which were already selected in pre-selection step #7.	See 6.3.2.2.3.

With:

Table 9 — Requirements for DRC pre-selection

#	Requirement	Applicability	Comment
1	DownmixId of DRC set matches the requested downmixId.	If a downmixId is requested	See 6.3.2.2.1.
2	Output channel layout of DRC set matches the requested layout.	If a target channel layout is requested	See 6.3.2.2.1.
3	Channel count of DRC set matches the requested channel count.	If a target channel count is requested	See 6.3.2.2.1.
4	The DRC set is not a “Fade-” or “Ducking-” only DRC set.	Always	DRC sets with “Fade” or “Ducking” effect are selected automatically. They are not subject to this selection process.
5	The number of DRC bands is supported.	Always	DRC sets that exceed the number of supported DRC bands are discarded. For time-domain DRC, the maximum is four bands.

Table 9 *(continued)*

6	Independent use of DRC set is permitted.	If the DRC set is not used in combination with another DRC set	DRC sets with a *noIndependentUse* flag value of 1 can only be used in combination with a second DRC set.
7	DRC sets that require EQ are only permitted if EQ is supported.	For all drcInstructionsUniDrcV1() payloads	If EQ is not supported, DRC sets with *requiresEq*==1 are discarded.
8	The range of the target loudness specified for a DRC set has to include the requested decoder target loudness.	If *drcSetTargetLoudnessPresent*==1 and no explicit peak information is available for that DRC set	See 6.3.2.2.2.
9	Clipping is minimized.	Except for DRC sets which were already selected in pre-selection step #8	See 6.3.2.2.3.

Page 12, 6.3

Replace all remaining occurences of #7 with #8 (3 times) and #8 with #9 (2 times).

Page 15, 6.3.2.2.2

Replace first paragraph with:

This pre-selection step addresses only DRC sets for which drcSetTargetLoudnessPresent equals one and for which no explicit peak information is available. From the DRC sets which match this criterion, only those are selected whose range defined by drcTargetLoudnessValueUpper/-Lower includes the requested decoder target loudness (targetLoudness). A range check shall include the upper boundary value and exclude the lower boundary value. Note that pre-selection step #8 is omitted for DRC sets selected in this step. Explicit peak information for a specific DRC set is available if at least one of the following results is TRUE:

Page 17, 6.3.3.3

Replace the third paragraph:

The program loudness values used in Table 12 shall be based on the following measurement systems (see Table A.37):

RMS_C

RMS_B

RMS_A

BS.1770-3

With:

The program loudness values used in Table 12 shall be based on the following measurement systems (see Table A.37):

RMS_C

RMS_B

RMS_A

BS.1770-4

Page 18, 6.3.4

Insert a paragraph after the first paragraph:

If the DRC tool is configured to support EQ and if there are still multiple DRC sets selected, select the ones that can be combined with an EQ of the requested 'EQ purpose' (see B.3.8.2). If no DRC set can be selected, select the ones that can be combined with an EQ of the default 'EQ purpose'. If still no DRC set can be selected, ignore this step.

Page 18, 6.3.4

Replace the paragraph:

If there are still multiple DRC sets selected, discard any DRC set that was selected in the pre-selection step dealing with drcSetTargetLoudness. If all selected DRC sets are discarded in this step, select the DRC set with the smallest drcSetTargetLoudnessValueUpper instead.

With:

If there are still multiple DRC sets selected, discard any DRC set that was selected in the pre-selection step dealing with drcSetTargetLoudness. If all selected DRC sets are discarded in this step, select the DRC set with the smallest drcSetTargetLoudnessValueUpper instead.

If there are still multiple DRC sets selected, select the DRC sets for which drcSetTargetLoudnessPresent equals one and whose range defined by drcSetTargetLoudnessValueUpper/-Lower includes the requested decoder target loudness. If multiple DRC sets are selected, select the DRC set with the smallest drcSetTargetLoudnessValueUpper.

Page 20, 6.4.5

Replace:

The assignment of a gain sequence to each channel. Channels using the same sequence are referred to as channel groups. The total number of groups is *nDrcChannelGroups* (see Table 14).

With:

The assignment of a gain set to each channel. Channels using the same gain set are referred to as channel groups. The total number of groups is nDrcChannelGroups (see Table 14).

Page 22, 6.4.5

Replace Table 14:

Table 14 — Derivation of drcChannelGroups from sequenceIndexes

```
uniqueIndex = {-10, -10, …};
k=0;
if ((drcSetEffect & (3<<10)) != 0) { /* Ducking */
  uniqueScaling = {-10, -10, …};
  for (i=0; i<channelCount; i++) {
    match = FALSE;
    idx = sequenceIndex[i];
    factor = duckingScaling[i];
    if (idx<0) channelGroupForChannel[i] = -1;  // means no DRC for this channel
    else {
      for (n=0; n<k; n++) {
        if ((uniqueIndex[n] == idx) && (uniqueScaling[n] == factor)) {
          match = TRUE;
          channelGroupForChannel[i] = n;
          break;
        }
      }
      if (match == FALSE) {
        uniqueIndex[k] = idx;
        uniqueScaling[k] = factor;
        channelGroupForChannel[i] = k;
        k++;
      }
    }
  }
}
else {
  for (i=0; i<channelCount; i++) {
    match = FALSE;
    idx = sequenceIndex[i];
    if (idx<0) channelGroupForChannel[i] = -1;
    else {
      for (n=0; n<k; n++) {
        if (uniqueIndex[n] == idx) {
          match = TRUE;
          channelGroupForChannel[i] = n;
          break;
        }
      }
      if (match == FALSE) {
        uniqueIndex[k] = idx;
        channelGroupForChannel[i] = k;
        k++;
      }
    }
```

Table 14 *(continued)*

```
  }
}
nDrcChannelGroups = k;
```

With:

Table 14 — Derivation of drcChannelGroups from gainSetIndices

```
uniqueIndex = {-10, -10, …};
k=0;
if ((drcSetEffect & (3<<10)) != 0) { /* Ducking */
  uniqueScaling = {-10, -10, …};
  for (i=0; i<channelCount; i++) {
    match = FALSE;
    idx = gainSetIndex[i];
    factor = duckingScaling[i];
    if (idx<0) channelGroupForChannel[i] = -1;  // means no DRC for this channel
    else {
      for (n=0; n<k; n++) {
        if ((uniqueIndex[n] == idx) && (uniqueScaling[n] == factor)) {
          match = TRUE;
          channelGroupForChannel[i] = n;
          break;
        }
      }
      if (match == FALSE) {
        uniqueIndex[k] = idx;
        uniqueScaling[k] = factor;
        channelGroupForChannel[i] = k;
        k++;
      }
    }
  }
}
else {
  for (i=0; i<channelCount; i++) {
    match = FALSE;
    idx = gainSetIndex[i];
    if (idx<0) channelGroupForChannel[i] = -1;
    else {
      for (n=0; n<k; n++) {
        if (uniqueIndex[n] == idx) {
          match = TRUE;
          channelGroupForChannel[i] = n;
          break;
        }
      }
      if (match == FALSE) {
        uniqueIndex[k] = idx;
```

Table 14 *(continued)*

```
            channelGroupForChannel[i] = k;
            k++;
        }
    }
  }
}
nDrcChannelGroups = k;
```

Page 22, 6.4.5

Replace the paragraph after Table 14:

The application of codes is expressed in Table 15 by the pseudo-functions decodeInitialGain(), decodeDeltaGain(), decodeTimeDelta(), and decodeSlope(). Differentially encoded values are then converted into absolute values according to Table 15. The decoded result is represented by the gain values *gDRC*[*g*][b][*k*], the time values *tDRC*[*g*][b][*k*], and the slope values *sDRC*[*g*][b][*k*] where *g* is the channel group index, *b* is the band index, and *k* is the node index. For linear interpolation, the slope values are set to 0. They will be neglected during interpolation. Time values are integer numbers relative to the beginning of the DRC frame in units of *deltaTmin*. The audio sample that coincides with the beginning of the DRC frame has a time value of *tDRC*=0.

With:

The application of codes is expressed in Table 15 by the pseudo-functions decodeInitialGain(), decodeDeltaGain(), decodeTimeDelta(), and decodeSlope(). Differentially encoded values are then converted into absolute values according to Table 15. The decoded result is represented by the gain values *gDRC*[*g*][b][*k*], the time values *tDRC*[*g*][b][*k*], and the slope values *sDRC*[*g*][b][*k*] where *g* is the channel group index, *b* is the band index, and *k* is the node index. The DRC gain sequences are assigned to channel groups and DRC bands using the information in the drcCoefficientsUniDrc() and drcInstructionsUniDrc() payloads or their V1 versions. For linear interpolation, the slope values are set to 0. They will be neglected during interpolation. Time values are integer numbers relative to the beginning of the DRC frame in units of *deltaTmin*. The audio sample that coincides with the beginning of the DRC frame has a time value of *tDRC*=0.

Page 24, 6.4.6

Replace the second paragraph:

As described in E.2.4 and E.4, there are several ways to adapt the DRC characteristics in the DRC tool decoder. These adjustments are applied to the decoded gain samples in the dB domain. The function toLinear() includes all necessary steps to generate a linear gain sample from the logarithmic value in dB (see Table 16). It contains an optional mapping function mapGain() (see Table 17) that supports modifications of the DRC gain values with the purpose of achieving a different compression characteristic than the one used in the encoder. The mapping is controlled by the index *drcCharacteristicTarget* that will select one of the custom decoder DRC characteristics if it is larger than 0. Otherwise, the encoder characteristic will not be replaced. A modified characteristic can be generated based on the encoder compression characteristic that is conveyed in the DRC configuration. Moreover, a compression and boost factor is supported to scale negative and positive gains, respectively. These factors have a value of 1.0, unless values in the range [0,1] are supplied by the user. Similarly, an encoder-controlled scaling is applied when *gainScalingPresent*==1 using the scale factors *attenuationScaling* and *amplificationScaling*. When ducking is active, the ducking gains in dB are scaled by the factor *duckingScaling,* if present. Note that the *duckingScaling* factors are conveyed in the drcInstructionsUniDrc() payload for the channel they are applied to, which is in contrast to the *bsSequenceIndex* channel assignment for the “Duck other”

effect. User supplied compression and boost factors shall be applied to all DRC sets except DRC sets with ducking, fading, or clipping effect.

With:

As described in E.2.4 and E.4, there are several ways to adapt the DRC characteristics in the DRC tool decoder. These adjustments are applied to the decoded gain samples in the dB domain. The function toLinear() includes all necessary steps to generate a linear gain sample from the logarithmic value in dB (see Table 16). It contains a mapping function mapGain() (see Table 17) that supports modifications of the DRC gain values with the purpose of achieving a different compression characteristic than the one used in the encoder. The mapping can be controlled by the host (*drcCharacteristicTarget* > 0) which selects one of the target characteristics based on an index. Otherwise, if specified in the drcInstructionsUniDrcV1() payload, a target characteristic is used and the DRC gain values are mapped to it using the mapGain() function. In that case, different characteristics or scalings can be applied independently to positive and negative gains (see Table 16). Otherwise, the encoder characteristic will not be replaced. A modified characteristic can be generated based on the encoder compression characteristic that is conveyed in the DRC configuration. Moreover, a compression and boost factor is supported to scale negative and positive gains, respectively. These factors have a value of 1.0, unless values in the range [0,1] are supplied by the user. Similarly, an encoder-controlled scaling is applied when *gainScalingPresent*==1 using the scale factors *attenuationScaling* and *amplificationScaling*. When ducking is active, the ducking gains in dB are scaled by the factor *duckingScaling,* if present. Note that the *duckingScaling* factors are conveyed in the drcInstructionsUniDrc() payload for the channel they are applied to, which is in contrast to the *bsGainSetIndex* channel assignment for the "Duck other" effect. User-supplied compression and boost factors shall be applied to all DRC sets except DRC sets with ducking, fading, or clipping effect.

Page 24, 6.4.6

Insert a new paragraph between the second and third paragraph:

In a drcCoefficientsUniDrcV1() payload, custom DRC characteristics can be defined to support more flexible gain modifications. These parametric characteristics can be used to describe the encoder-side DRC and the target characteristic. If a target characteristic is defined, it shall be applied after inverting the encoder-side characteristic. Hence, for the most general use, the encoder-side characteristic shall be invertible, i.e. have either a negative or positive slope for the entire gain range. Moreover, if the target characteristic has a section with constant gain, the encoder-side characteristic may also have constant gain in those sections. The parametric characteristics are computed as indicated in Table AMD1.2 and Table AMD1.3. Further details can be found in E.4.

Page 24, 6.4.6

Replace Table 16:

Table 16 — Conversion of a DRC gain sample and associated slope from dB to linear domain

```
toLinear (gainDb, slopeDb) {
  SLOPE_FACTOR_DB_TO_LINEAR = 0.1151f;      /* ln(10) / 20 */
  EFFECT_BIT_CLIPPING = 0x0100;             /* drcSetEffect 9 (Clip.Prev.) */
  EFFECT_BIT_FADE     = 0x0200;             /* drcSetEffect 10 (Fade) */
  EFFECT_BITS_DUCKING = 0x0400 | 0x0800;    /* drcSetEffect 11 or 12 (Ducking) */
  gainRatio = 1.0;
  if (((drcSetEffect & EFFECT_BITS_DUCKING) == 0) &&
      (drcSetEffect != EFFECT_BIT_FADE) &&
      (drcSetEffect != EFFECT_BIT_CLIPPING)) {
    if ((drcCharacteristicTarget > 0)    && (gainDb != 0.0)){
        gainRatio = mapGain(gainDb) / gainDb;
```

Table 16 *(continued)*

```
    }
    if (gainDb < 0.0) {
     gainRatio *= compress;
    }
    else {
     gainRatio *= boost;
    }
  }
  if (gainScalingPresent) {
    if (gainDb < 0.0) {
     gainRatio *= attenuationScaling;
    }
    else {
     gainRatio *= amplificationScaling;
    }
  }
  if (duckingScalingPresent && (drcSetEffect & EFFECT_BITS_DUCKING)) {
    gainRatio *= duckingScaling;
  }
  gainLin = pow(2.0, gainRatio * gainDb / 6.0);
  slopeLin = SLOPE_FACTOR_DB_TO_LINEAR * gainRatio * gainLin * slopeDb;
  if (gainOffsetPresent) {
     gainLin *= pow(2, gainOffset/6.0);
  }
  /* The only drcSetEffect is "clipping prevention" */
  if (limiterPeakTargetPresent && (drcSetEffect == EFFECT_BIT_CLIPPING)) {
     gainLin *= pow(2, max(0.0, -limiterPeakTarget-loudnessNormalizationGainDb
                                -loudnessNormalizationGainModificationDb)/6.0);
     if (gainLin >= 1.0) {
        gainLin = 1.0;
        slopeLin = 0.0;
     }
  }
  return (gainLin, slopeLin);
}
```

With:

Table 16 — Conversion of a DRC gain sample and associated slope from dB to linear domain (slopeIsNegative == 1 if the source DRC characteristic has a negative slope)

```
toLinear (gainDb, slopeDb) {
  SLOPE_FACTOR_DB_TO_LINEAR = 0.1151f;          /* ln(10) / 20 */
  EFFECT_BIT_CLIPPING = 0x0100;                  /* drcSetEffect 9 (Clip.Prev.) */
  EFFECT_BIT_FADE     = 0x0200;                 /* drcSetEffect 10 (Fade) */
  EFFECT_BITS_DUCKING = 0x0400 | 0x0800;        /* drcSetEffect 11 or 12 (Ducking) */
  gainRatio = 1.0;
  if (((drcSetEffect & EFFECT_BITS_DUCKING) == 0) &&
      (drcSetEffect != EFFECT_BIT_FADE) &&
      (drcSetEffect != EFFECT_BIT_CLIPPING)) {
    if ((drcCharacteristicTarget > 0)  && (gainDb != 0.0)){
```

Table 16 *(continued)*

```
        gainRatio = mapGain(gainDb) / gainDb;   /* target characteristic from host */
    }
    else if (drcCoefficientsUniDrcV1Present == 1) {
      if (((gainDb > 0.0) && (slopeIsNegative == 1)) ||
          ((gainDb < 0.0) && (slopeIsNegative == 0))) {
        if (targetCharacteristicLeftPresent == 1) {
          gainRatio = mapGain(gainDb) / gainDb; /* target characteristic in payload */
        }
      }
      else if (((gainDb < 0.0) && (slopeIsNegative == 1)) ||
               ((gainDb > 0.0) && (slopeIsNegative == 0))) {
        if (targetCharacteristicRightPresent == 1) {
          gainRatio = mapGain(gainDb) / gainDb; /* target characteristic in payload
*/        }
      }
    }
    if (gainDb < 0.0) {
     gainRatio *= compress;
    }
    else {
     gainRatio *= boost;
    }
  }  if (gainScalingPresent) {
    if (gainDb < 0.0) {
     gainRatio *= attenuationScaling;
    }
    else {
     gainRatio *= amplificationScaling;
    }
  }
  if (duckingScalingPresent && (drcSetEffect & EFFECT_BITS_DUCKING)) {
    gainRatio *= duckingScaling;
  }
  gainLin = pow(2.0, gainRatio * gainDb / 6.0);
  slopeLin = SLOPE_FACTOR_DB_TO_LINEAR * gainRatio * gainLin * slopeDb;

  if (gainOffsetPresent) {
     gainLin *= pow(2, gainOffset/6.0);
  }
  /* The only drcSetEffect is "clipping prevention" */
  if (limiterPeakTargetPresent && (drcSetEffect == EFFECT_BIT_CLIPPING)) {
     gainLin *= pow(2, max(0.0, -limiterPeakTarget-loudnessNormalizationGainDb
                                -loudnessNormalizationGainModificationDb)/6.0);
     if (gainLin >= 1.0) {
        gainLin = 1.0;
        slopeLin = 0.0;
     }
  }

  return (gainLin, slopeLin);
}
```

Page 25, 6.4.6

Insert the following two tables after Table 17:

Table AMD1.2 — Pseudo code to compute the DRC gain based on the input level in dB (characteristicFormat==0)

```
drcInputLoudnessTarget = -31;
compressorIO_sigmoidLeft(inLevelDb) {
  if (inLevelDb > drcInputLoudnessTarget) {
    printf("ERROR!");
    return (0.0);
  }
  tmp = (drcInputLoudnessTarget - inLevelDb) * ioRatioLeft;
  if (expLeft < INF) {                                  /* INF means infinity */
    outGainDb = tmp / pow(1 + pow(tmp/gainDbLeft, expLeft), 1/expLeft);
  } else {
    outGainDb = tmp;
  }
  if (flipSignLeft == 0) return outGainDb;
  else return -outGainDb;
}
compressorIO_sigmoidRight(inLevelDb) {
  if (inLevelDb < drcInputLoudnessTarget) {
    printf("ERROR!");
    return (0.0);
  }
  tmp = (drcInputLoudnessTarget - inLevelDb) * ioRatioRight;
  if (expRight < INF) {                                  /* INF means infinity */
    outGainDb = tmp / pow(1 + pow(tmp/gainDbRight, expRight), 1/expRight);
  } else {
    outGainDb = tmp;
  }
  if (flipSignRight == 0) return outGainDb;
  else return -outGainDb;
}
```

Table AMD1.3 — Pseudo code to compute the DRC gain based on the input level in dB (characteristicFormat==1)

```
drcInputLoudnessTarget = -31
compressorIO_nodesLeft(inLevelDb) {
  if (inLevelDb > drcInputLoudnessTarget) {
    printf("ERROR!");
    return (0.0);
  }
  nodeLevel[0] = drcInputLoudnessTarget;
  nodeGain[0] = 0.0;
  for (n=1; n<=characteristicNodeCount; n++) {
    nodeLevel[n] = nodeLevel[n-1] - nodeLevelDelta[n];
  }
  for (n=1; n<=characteristicNodeCount; n++) {
    if ((inLevelDb <= nodeLevel[n-1]) && (inLevelDb > nodeLevel[n])) {
```

Table AMD1.3 *(continued)*

```
      w = (nodeLevel[n]- inLevelDb)/(nodeLevel[n]-nodeLevel[n-1]);
      return (w * nodeGain[n-1] + (1.0-w) * nodeGain[n]);
    }
  }
  return (nodeGain[characteristicNodeCount]);
}
compressorIO_nodesRight(inLevelDb) {
  if (inLevelDb < drcInputLoudnessTarget) {
    printf("ERROR!");
    return (0.0);
  }
  nodeLevel[0] = drcInputLoudnessTarget;
  nodeGain[0] = 0.0;
  for (n=1; n<=characteristicNodeCount; n++) {
    nodeLevel[n] = nodeLevel[n-1] + nodeLevelDelta[n];
  }
  for (n=1; n<=characteristicNodeCount; n++) {
    if ((inLevelDb >= nodeLevel[n-1]) && (inLevelDb < nodeLevel[n])) {
      w = (nodeLevel[n]- inLevelDb)/(nodeLevel[n]-nodeLevel[n-1]);
      return (w * nodeGain[n-1] + (1.0-w) * nodeGain[n]);
    }
  }
  return (nodeGain[characteristicNodeCount]);
}
```

Page 29, 6.4.9

Replace the first paragraph:

To reduce bitrate peaks, a so-called node reservoir can be employed. The node reservoir can be only applied in the default delay mode, where the DRC gain is applied with a delay of one DRC frame (see 6.4.8). The node reservoir mechanism shifts nodes from the current frame into the subsequent frame (see Figure 5). In order to keep the decoding delay constant, it is not allowed to shift the first node of a frame since in default delay mode, the first node is needed to fully decode the previous frame. Thus, the maximum number of nodes that can be shifted to the subsequent frame is limited to *nNodes-1* nodes.

With:

To reduce bitrate peaks, a so-called “node reservoir” can be employed. The node reservoir can be only applied in the default delay mode, where the DRC gain is applied with a delay of one DRC frame (see 6.4.8). In addition, the *fullFrame* flag shall be set to 0. The node reservoir mechanism shifts nodes from the current frame into the subsequent frame (see Figure 5). In order to keep the decoding delay constant, it is not allowed to shift the first node of a frame since in default delay mode, the first node is needed to fully decode the previous frame. Thus, the maximum number of nodes that can be shifted to the subsequent frame is limited to *nNodes-1* nodes.

Page 30, 6.4.10

Insert a new subclause after 6.4.10:

6.4.11 Dynamic equalization

6.4.11.1 Overview

Shaping filters are used to dynamically modify the audio signal spectrum as the DRC gain changes, i.e. the filters provide a kind of dynamic equalization. Figure AMD1.1 shows the shaping filters for time-domain processing that includes filters to cut and boost the low and high frequency range. The filter coefficients are dynamically adapted depending on the final DRC gain value after interpolation. If the gain value is below 0 dB, the filters for cutting have a flat frequency response. In contrast, when the gain value is above 0 dB, the filters for boosting have a flat frequency response. The filter coefficient adaptation is controlled by the parameters for corner frequency and filter strength.

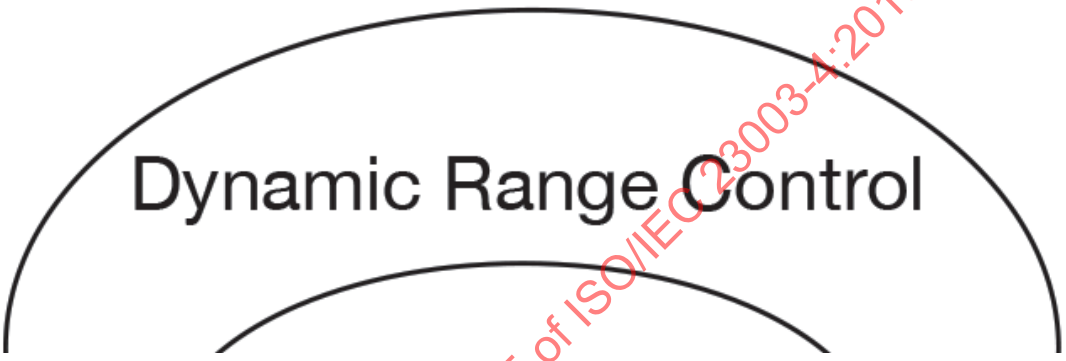

Figure AMD1.1 — Bank of spectral shaping filters

Shaping filters are only applied for single-band DRCs and if the DRC operates in the time domain. They are ignored in the QMF domain. If a similar effect is desired for operation in the QMF domain, it is recommended to use a multi-band DRC instead. Please note that the application of shaping filters introduces a phase shift of the audio signal. Hence, if the shaping filters of the channel groups differ, the output channel groups may not be phase aligned anymore. Therefore, it is recommended to apply the same shaping filter bank to all channel groups, unless the audio signal groups are uncorrelated.

Each filter has one adaptive coefficient, which is dynamically computed based on the linear DRC gain value g_{DRC} (after interpolation) as indicated by the formula for y_1. The linear gain value is first warped according to the pseudo code for warpGain() in Table AMD1.4.

$$g_{\mathrm{warped}} = \mathrm{warpGain}\left(g_{\mathrm{DRC}}, \mathrm{mode}\right)$$

$$y_1 = \begin{cases} x_1 & ; \quad g_{\mathrm{warped}} \le 0 \\ x_1 + \left(y_{1,\mathrm{bound}} - x_1\right)\dfrac{g_{\mathrm{warped}}}{g_{\mathrm{warped,max}}} & ; \quad 0 < g_{\mathrm{warped}} < g_{\mathrm{warped,max}} \\ y_{1,\mathrm{bound}} & ; \quad \mathrm{else} \end{cases}$$

with $y_{1,\mathrm{bound}}$ as defined in Table AMD1.47 and Table AMD1.48, and

$$g_{\mathrm{warped,max}} = \mathrm{warpGain}\left(g_{\mathrm{DRC,max}}, \mathrm{'cut'}\right) \text{ with } g_{\mathrm{DRC,max}} = 10^{32/20}$$

The warping function is defined in Table AMD1.4 using the gain offset parameter g_{offset} as defined in Table AMD1.49 and Table AMD1.50.

Table AMD1.4 — Pseudo code of warpGain() function

```
warpGain(gDrc, mode) {
    switch(mode) {
        case 'cut':   /* LF or HF cut filter */
               if (gDrc <= 1.0) return (-1.0);
               else return (gDrc-1)/(gDrc-1+gOffset);
        case 'boost':   /* LF or HF boost filter */
               if (gDrc >= 1.0) return (-1.0);
               else return (1-gDrc)/(1+gDrc*(gOffset-1));
    }
}
```

6.4.11.2 Adaptation of shaping filters

All shaping filters are adapted at the first audio sample of each output audio frame and whenever the DRC gain change compared to the previous adaptation exceeds the threshold according to the following formula:

$$\mathrm{abs}\left(g_{\mathrm{DRC}} - g_{\mathrm{DRC,last}}\right) > 0.0001\, g_{\mathrm{DRC,last}}$$

where $g_{\mathrm{DRC,last}}$ is the DRC gain value when the shaping filters were last updated. On initialization, the value is set to $g_{\mathrm{DRC,last}} = 1.0$.

6.4.11.3 Low-frequency shaping filters

Each low-frequency filter is a first order IIR filter with real coefficients of the form:

$$H_{\mathrm{LF}}(z) = \frac{1 + b_1 z^{-1}}{1 + a_1 z^{-1}}$$

The coefficient x_1 is defined depending on the radius, r, in Table AMD1.51 as

$$x_1 = -r$$

The filter coefficients for the LF filters are assigned as shown in Table AMD1.5.

Table AMD1.5 — Assignment of filter coefficients for LF shaping filters

Filter type	Filter coefficients	
LF cut filter	$a_1 = x_1$	$b_1 = y_1$
LF boost filter	$a_1 = y_1$	$b_1 = x_1$

6.4.11.4 High-frequency shaping filters

Each high-frequency filter is a second order IIR filter with real coefficients of the form:

$$H_{\mathrm{HF}}(z) = g_{\mathrm{norm}} \frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \qquad g_{\mathrm{norm}} = \frac{1 + a_1 + a_2}{1 + b_1 + b_2}$$

The corner frequency of the filter depends on the audio sample rate f_s and the normalized corner frequency $f_{c,\mathrm{norm}}$:

$$f_c = f_{c,\mathrm{norm}} f_s$$

The fixed coefficients for both, the HF cut and boost filter depend on the corner frequency $f_{c,\mathrm{norm}}$ and the pole/zero radius parameter r (see Table AMD1.52).

$$x_1 = -2r\cos\left(2\pi f_{c,\mathrm{norm}}\right)$$

$$x_2 = r^2$$

$$y_2 = x_2$$

The filter coefficients for the HF filters are assigned as shown in Table AMD1.6.

Table AMD1.6 — Assignment of filter coefficients for HF shape filters

Filter type	Filter coefficients			
HF cut filter	$a_1 = x_1$	$a_2 = x_2$	$b_1 = y_1$	$b_2 = y_2$
HF boost filter	$a_1 = y_1$	$a_2 = y_2$	$b_1 = x_1$	$b_2 = x_2$

Page 40, 6.6

Add new subclauses before 6.6 and adjust the subsequent clause numbers accordingly:

6.6 Generation of DRC gain values at the decoder

In addition to dynamic DRC gains included in the uniDrcGain() payload, the DRC tool alternatively allows to generate time-varying DRC gain values at the decoder based on transmitted parametric DRC configurations. The parametric DRC configurations are conveyed as an extension to the static uniDrcConfig() payload (see Table 50 and Table A.12).

6.6.1 Overview

Parametric DRC payloads are transmitted in the uniDrcConfigExtension() payload. The following logical blocks are available:

— drcCoefficientsParametricDrc()

— parametricDrcInstructions()

— parametricDrcTypeFeedForward()

— parametricDrcTypeLimiter()

Except for the drcCoefficientsParametricDrc() block, multiple instances of a logical block can appear in the bitstream. For the definition of a DRC set, a drcInstructionsUniDrc() block can refer to a DRC gain set defined in drcCoefficientsUniDrc() and/or to a DRC gain set defined in drcCoefficientsParametricDrc(). If a drcCoefficientsUniDrc() and a drcCoefficientsParametricDrc() block for the same *drcLocation* are present, the first *gainSetIndex* referring to drcCoefficientsParametricDrc() starts with an offset of the *gainSetCount* defined in drcCoefficientsUniDrc(). There is no distinction between parametric DRCs and DRCs based on dynamic gain values in the DRC selection process, because the selection is based on the drcInstructionsUniDrc() blocks. Note that the specification in this subclause equally holds in the context of the extended payloads downmixInstructionsV1(), drcCoefficientsUniDrcV1(), drcInstructionsUniDrcV1() and loudnessInfoV1().

For parametric DRCs that require a make-up gain, the *gainOffset* field within drcInstructionsUniDrc() is employed.

Detailed information on each logical block can be found in 6.6.2. Algorithmic details are specified in 6.6.3.

6.6.2 Description of logical blocks

6.6.2.1 drcCoefficientsParametricDrc()

A drcCoefficientsParametricDrc() block includes top-level fields for the definition of parametric DRC gain sets. The *drcLocation* field is required for finding the first *gainSetIndex* referring to drcCoefficientsParametricDrc() (see Table AMD1.53). The *parametricDrcFrameSize* field indicates the processing frame size. The maximum delay that can be incurred by any parametric DRC set in the stream can be specified in *bsParametricDelayMax.* Parametric DRCs can be reset at the decoder by using the *resetParametricDrc* field (e.g. in streaming scenarios). A reset shall initialize all internal algorithmic states to their default value. *parametricDrcGainSetCount* signals the number of independent parametric DRC configurations in the bitstream.

Each parametric DRC gain set refers to a parametricDrcInstructions() block by a unique *parametricDrcId*. *sideChainConfigType* defines the side-chain configuration of a parametric DRC gain set: For a value of 1, *levelEstimChannelWeightFormat* defines whether a simple channel map (*addChannel[]*) or individual channel weights (*channelWeight[]*) are provided for the definition of the side-chain input signal. For all other values, the side-chain input signal is defined by the *drcChannelGroup* the parametric DRC gain set is assigned to within a drcInstructionsUniDrc() block. If a drcInstructionsUniDrc() block with *downmixId*=X refers to a DRC gain set with present but non-matching *downmixId*=Y, *sideChainConfigType* shall be set to 0. Note that for DRC sets with drcSetEffect “Duck other” and sideChainConfigType not equal to 1, the side-chain signal is defined by the *drcChannelGroup* the ducking gain set is associated with.

For normalization of the DRC side-chain input level, each parametric DRC gain set can define an individual *drcInputLoudness* value. Alternatively, it can be selected from an applicable loudnessInfo() structure. The applicable loudnessInfo() structure is selected according to 6.6 (Loudness normalization), wherein the requested *drcSetId* shall be zero (audio-signal without DRC processing) and the requested *downmixId* shall match the *downmixId* of the processed DRC set. Loudness values matching the processed *drcChannelGroup* (or to a customized side-chain input signal) shall be preferred if present (e.g. for MPEG-H 3D Audio).

6.6.2.2 parametricDrcInstructions()

The *parametricDrcId* field represents a unique identifier for each parametricDrcInstructions() block. Each parametric DRC can define an individual DRC look-ahead delay if required. Note that for constant delay audio decoders, the definition of an application-specific maximum look-ahead delay (*parametricDrcLookAheadMax*) for parametric DRCs is required. For such applications, it is recommended to include the *bsParamtericDrcDelayMax* parameter in the bitstream that specifies a constant decoder delay that is large enough to cover the lookahead delay of any parametric DRC set in the stream. During the decoding process, a delay shall be applied to compensate for any difference between *parametricDrcLookAheadMax* and *parametricDrcLookAhead*. For *parametricDrcLookAhead* > *parametricDrcLookAheadMax*, the DRC tool shall set *parametricDrcLookAhead* = *parametricDrcLookAheadMax*. Note that for simultaneously applied DRC sets, *parametricDrcLookAheadMax* should account for all parametric DRC instances in the chain.

The *parametricDrcPresetId* index can be used to efficiently select a suitable *parametricDrcType* and corresponding parametric DRC settings (see Table AMD1.62)

If *parametricDrcPresetId* is not present, *parametricDrcType* indicates the type of parametric DRC that shall be used. Dependent on the selected type, further payloads follow in the bitstream that allow configuring specific parameters of a DRC algorithm:

— *parametricDrcType*==0x0: Parameters defined by parametricDrcTypeFeedForward() block (see 6.6.3.1 for algorithmic details).

— *parametricDrcType*==0x1: Parameters defined by parametricDrcTypeLimiter() block (see 6.6.3.2 for algorithmic details).

The parametricDrcInstructions() block can include future extension payloads by definition of new *parametricDrcType* values. In the event that a *parametricDrcType* value is received that is not supported by a decoder implementation, the DRC tool parser shall read and discard the bits (*otherBit*) of the unknown *parametricDrcType* payload. The corresponding parametric DRC instance shall be disabled in that case.

6.6.2.3 parametricDrcTypeFeedForward()

The parametricDrcTypeFeedForward() block represents parameters of a standard feed-forward DRC design. The *levelEstimKWeightingType* field indicates which filters shall be applied before level estimation. *levelEstimIntegrationTime* controls the integration time used for level estimation in multiples of *parametricDrcFrameSize*. The gain-mapping curve of the parametric DRC can be either set by a *drcCharacteristic* index according to ISO/IEC 23001-8 or by definition of individual curve segments. Note that parameters such as maximum-boost or maximum-compress can be realized within the DRC curve parameterization. Various gain-smoothing parameters can be customized if required. See 6.6.3.1 for further algorithmic details. Note that the presence of a peak limiter is strongly recommended for this parametric DRC design since it does not include means for generation of clipping prevention gains for the intended target loudness. Note that a parametric DRC instance of type *parametricDrcType*=0x0 can also be combined with a parametric DRC instance of type *parametricDrcType*=0x1 by using the *dependsOnDrcSet* field (see D.2.8).

6.6.2.4 parametricDrcTypeLimiter()

The parametricDrcTypeLimiter() block represents parameters of a time domain peak limiter which is designed to prevent clipping of the time domain output signal. The *parametricLimThreshold* field indicates the limiting threshold. The *parametricLimAttackTime* and *parametricLimReleaseTime* parameters control the gain smoothing behavior of the limiting algorithm. *parametricLimAttackTime* is implicitly provided by the *parametricDrcLookAhead* field in the parametricDrcInstructions() block. *parametricLimReleaseTime* can be explicitly conveyed by using the *bsParametricLimReleaseTime* field. See 6.6.3.2 for further algorithmic details. Note that the parameters of parametricDrcTypeLimiter() shall not have any influence on the DRC pre-selection #8 according to 6.3.2.2.3. Vice versa, the *limiterPeakTarget* parameter does not control this peak limiter. Further note that gain manipulation as outlined in 6.4.6 is not permitted for parametricDrcType=0x1.

6.6.3 Algorithmic details

6.6.3.1 Parametric DRC of type PARAM_DRC_TYPE_FF

The parametric DRC of type PARAM_DRC_TYPE_FF (0x0) is a single-band feed-forward design. A block diagram is depicted in Figure AMD1.2.

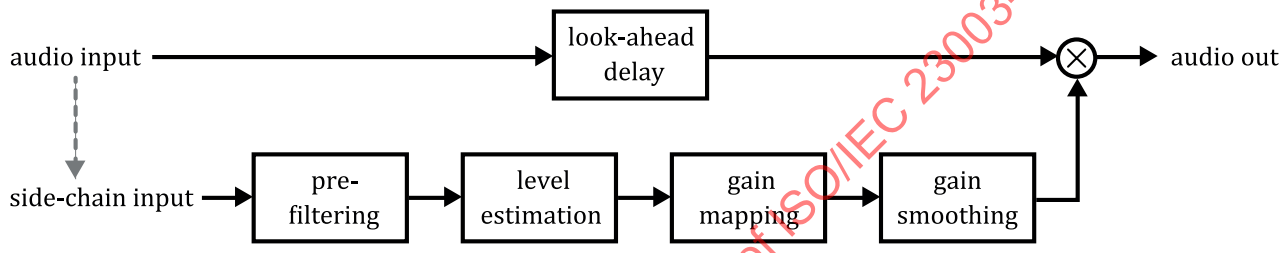

Figure AMD1.2 — Single-band feed-forward design of parametricDrcType==0x0

Dependent on *levelEstimKWeightingType*, the side-chain signal is filtered by two independent pre-filters according to ITU-R BS.1770-4. Each filter can be implemented by a second order section with the following transfer function:

$$H(z)=\frac{b_0+b_1z^{-1}+b_2z^{-2}}{a_0+a_1z^{-1}+a_2z^{-2}}$$

Note that the filter coefficients according to ITU-R BS.1770-4 hold for a sampling rate of 48 kHz. Audio decoders that operate at other sampling rates will require different coefficient values, which should be chosen to provide the same frequency response that the specified filters provide at 48 kHz. For audio decoders that operate in the sub-band domain, the filters shall be realized by appropriate sub-band weighting. Note that *parametricDrcFrameSize* shall be a multiple of the sub-band sampling interval. If the processing domain is not deterministic at the encoder, *parametricDrcFrameSize* shall be configured based on the sub-band sampling interval.

After pre-filtering, the level is estimated either in the time domain or sub-band domain according to Table AMD1.7. Note that the energy of multiple parametric DRC processing frames (*parametricDrcFrameSize*) has to be accumulated dependent on *levelEstimIntegrationTime*.

Table AMD1.7 —Level estimation for time domain and sub-band domain processing for one parametric DRC frame (parametricDrcType==0x0)

```
/* level estimation for time domain processing */
estimLevelTimeDomain
{
     drcInputLoudnessTarget = -31;
     /* accumulate energy + normalize */
     levelLin = 0;
     for(c=0; c<nChannels; c++) {
```

Table AMD1.7 *(continued)*

```
        for(t=0; t<levelEstimIntegrationTime; t++) {
           levelLin += levelEstimChannelWeight[c] * pow(audioSampleIn[c][t],2);
        }
      }
      levelLin = levelLin / levelEstimIntegrationTime;
      /* compute level */
      if (levelLin < 1e-10) levelLin = 1e-10;
      if (levelEstimKWeightingType == 2) { // pre-filter ON
        levelDb = -0.691 + 10*log10(levelLin) + 3;
      } else {
        levelDb = 10*log10(levelLin) + 3;
      }
      levelDb = levelDb - drcInputLoudness + drcInputLoudnessTarget;
      return levelDb;
}
/* level estimation for sub-band domain processing. */
estimLevelSubbandDomain
{
      drcInputLoudnessTarget = -31;

      /* accumulate energy + normalize */
      /* levelEstimIntegrationTimeSb is the number of sub-band samples that fit into …   */
         /* … levelEstimIntegrationTime */
      levelLin = 0;
      for(c=0; c<nChannels; c++) {
         for(s=0; s<nDecoderSubbands; s++) {
            for(m=0; m<levelEstimIntegrationTimeSb; m++) {
                levelLin += levelEstimChannelWeight[c] *
                         pow(abs(audioSampleSbInWeighted[c][s][m]),2);
            }
         }
      }
      levelLin = levelLin / (nDecoderSubbands * levelEstimIntegrationTimeSb);
      /* compute level */
      if (levelLin < 1e-10) levelLin = 1e-10;
      if (levelEstimKWeightingType == 2) { // pre-filter ON
        levelDb = -0.691 + 10*log10(levelLin) + 3;
      } else {
        levelDb = 10*log10(levelLin) + 3;
      }
      levelDb = levelDb - drcInputLoudness + drcInputLoudnessTarget;
      return levelDb;
}
```

Next, the estimated level is mapped to a gain value according to Table AMD1.8, which also includes an initialization routine based on the received curve segments. An illustration of the curve parameterization for 5 curve nodes is depicted in Figure AMD1.6.

Table AMD1.8 —Gain mapping for one parametric DRC frame (parametricDrcType==0x0)

```
mapLevelToGain
{
    /* initialization (only first frame) */
    initNodeLevel();
    /* determination of segment */
    for(c=0; c<nodeCount; c++) {
      if (levelDb <= nodeLevel[c]) {
         break;
      }
    }
    /* gain mapping */
    if (c == 0) {
      gainDb = nodeGain[c];
    } else if (c == nodeCount) {
      gainDb = nodeGain[c-1] - levelDb + nodeLevel[c-1];
    } else {
      gainDb = nodeGain[c] +
                    (levelDb - nodeLevel[c]) /
                    (nodeLevel[c-1] - nodeLevel[c]) *
                    (nodeGain[c-1] - nodeGain[c]);
    }
    return gainDb;
}
initNodeLevel
{
    /* initialization (only for drcCurveDefinitionType==1) */
    for(c=0; c<nodeCount; c++) {
      if (c == 0) {
        nodeLevel[c] = nodeLevelInitial;
      }
      else {
        nodeLevel[c] = nodeLevel[c-1] + nodeLevelDelta[c];
      }
    }
    return nodeLevel;
}
```

Next, temporal smoothing is applied to the gains of consecutive parametric DRC frames. A pseudo-function of the gain smoothing is listed in Table AMD1.9.

Table AMD1.9 —Gain smoothing for one parametric DRC frame (parametricDrcType==0x0)

```
smoothGain(gainDb, gainSmoothDbPrevious,
           levelDb, levelSmoothDbPrevious)
{
     /* initialization (only first frame) */
     initSmoothingParameters();
     /* get alpha */
     levelDelta = levelDb-levelSmoothDbPrevious;
     if (gainDb < gainSmoothDbPrevious) {
        /* attack */
        if (levelDelta > gainSmoothAttackThreshold) {
           alpha = alphaAttackFast;
        } else {
           alpha = alphaAttackSlow;
        }
     } else {
        /* release */
        if (levelDelta < -gainSmoothReleaseThreshold) {
           alpha = alphaReleaseFast;
        } else {
           alpha = alphaReleaseSlow;
        }
     }
     /* smooth gain and level */
     if (gainDb < gainSmoothDbPrevious || holdCounter == 0) {
        levelSmoothDb = (1-alpha) * levelSmoothDbPrevious + alpha * levelDb;
        gainSmoothDb = (1-alpha) * gainSmoothDbPrevious + alpha * gainDb;
     }
     /* holdCounter */
     if (holdCounter > 0) {
        holdCounter = holdCounter - 1;
     }
     if (gainDb < gainSmoothDb) {
        holdCounter = holdOffCount;
     }
     return (gainSmoothDb,levelSmoothDb);
}
initSmoothingParameters
{
     alphaAttackFast  = 1 - exp(-1.0 * parametricDrcFrameSize /
                          (gainSmoothAttackTimeFast * fs * 0.001));
     alphaReleaseFast = 1 - exp(-1.0 * parametricDrcFrameSize /
                          ( (gainSmoothReleaseTimeFast * fs * 0.001));
     alphaAttackSlow  = 1 - exp(-1.0 * parametricDrcFrameSize /
                          (gainSmoothAttackTimeSlow * fs * 0.001));
     alphaReleaseSlow = 1 - exp(-1.0 * parametricDrcFrameSize /
                          (gainSmoothReleaseTimeSlow * fs * 0.001));
     holdOffCount = floor(gainSmoothHoldOff * 0.0053 * fs / parametricDrcFrameSize);
     levelSmoothDbPrevious = -135;
```

Table AMD1.9 *(continued)*

```
    gainSmoothDbPrevious = 0;
    holdCounter = 0;
}
```

Finally, the smoothed gains are interpolated and applied to the audio signal in the time domain or sub-band domain according to Table AMD1.10. Note that gain modifications as outlined in 6.4.6 can be equally applied to parametric DRC gains sets that were generated with *parametricDrcType*=0x0.

parametricDrcLookAhead indicates the delay that shall be applied to the audio signal before the computed gains are applied. The delay can be used to compensate for delay that is introduced to the computed gains by level estimation (*levelEstimIntegrationTime*), gain smoothing (e.g. *gainSmoothAttackTimeFast*) and linear interpolation during gain application.

Table AMD1.10 — Gain application for time domain and sub-band domain processing for one DRC frame (parametricDrcType==0x0)

```
applyGainTimeDomain(audioSampleIn, gainSmoothDb, gainSmoothPrevious)
{
      if (parametricDrcLookAheadPresent == 1) {
         drcLookAheadSamples = parametricDrcLookAhead * 0.001 * fs;
      }
      else {
         drcLookAheadSamples = parametricDrcLookAheadDefault * 0.001 * fs;
      }
      delayAudio(audioFrame, drcLookAheadSamples);
      numDrcSubframes = drcFrameSize/parametricDrcFrameSize;
      for(dsf=0; dsf<numDrcSubframes; dsf++)
      {
          gainInDb = gainSmoothDb[dsf];
          gainInLin = toLinearParametricDrc(gainInDb);
          gainInDiff = gainInLin - gainSmoothPrevious;
          for (t=0; t<parametricDrcFrameSize; t++)
          {
            audioSampleOut[dsf*parametricDrcFrameSize + t] =
              audioSampleIn[dsf*parametricDrcFrameSize + t] *
              ( gainSmoothPrevious + (t + 1) / parametricDrcFrameSize * gainInDiff );
          }
          gainSmoothPrevious = gainInLin;
      }
      return (audioSampleOut, gainSmoothPrevious);
}
applyGainSubbandDomain(audioSampleSbIn, gainSmoothDb, gainSmoothPrevious)
{
      if (parametricDrcLookAheadPresent == 1) {
          drcLookAheadSamples = parametricDrcLookAhead * 0.001 * fs;
      }
      else {
          drcLookAheadSamples = parametricDrcLookAheadDefault * 0.001 * fs;
      }
      L = drcFrameSize/drcFrameSizeSb; /* downsampling factor (hopsize) */
      delayAudioSb(audioFrameSb, floor(drcLookAheadSamples/L));
      cnt = 0;
```

Table AMD1.10 *(continued)*

```
      m = 0;
      numDrcSubframes = drcFrameSize/parametricDrcFrameSize;
      for(dsf=0; dsf<numDrcSubframes; dsf++)
      {
          gainInDb = gainSmoothDb[dsf];
          gainInLin = toLinearParametricDrc(gainInDb);
          gainInDiff = gainInLin - gainSmoothPrevious;
          for (t=0; t<parametricDrcFrameSize; t++)
          {
            if (cnt == L-1)
            {
              for (s=0; s<nDecoderSubbands; s++)
              {
                audioSampleSbOut[s][m] = audioSampleSbIn[s][m] *
               ( gainSmoothPrevious +  (t + 1) / parametricDrcFrameSize * gainInDiff );
              }
              m++;
              cnt = -1;
            }
            cnt++;
          }
          gainSmoothPrevious = gainInLin;
       }
      return (audioSampleSbOut, gainSmoothPrevious);
}
toLinearParametricDrc(gainDb) {
     EFFECT_BIT_CLIPPING = 0x0100;          /* drcSetEffect 9 (Clip.Prev.) */
     EFFECT_BIT_FADE     = 0x0200;          /* drcSetEffect 10 (Fade) */
     EFFECT_BITS_DUCKING = 0x0400 | 0x0800; /* drcSetEffect 11 or 12 (Ducking) */
     gainRatio = 1.0;
     if (((drcSetEffect & EFFECT_BITS_DUCKING) == 0) &&
        (drcSetEffect != EFFECT_BIT_FADE) &&
        (drcSetEffect != EFFECT_BIT_CLIPPING)) {
        if (gainDb < 0.0) {
           gainRatio *= compress;
        }
        else {
           gainRatio *= boost;
        }
     }
     if (gainScalingPresent) {
        if (gainDb < 0.0) {
           gainRatio *= attenuationScaling;
        }
        else {
           gainRatio *= amplificationScaling;
        }
     }
     if (duckingScalingPresent && (drcSetEffect & EFFECT_BITS_DUCKING))
        gainRatio *= duckingScaling;
```

Table AMD1.10 *(continued)*

```
    }
    gainLin = pow(2.0, gainRatio * gainDb / 6.0);
    if (gainOffsetPresent) {
       gainLin *= pow(2.0, gainOffset / 6.0);
    }
    return (gainLin);
}
```

6.6.3.2 Parametric DRC of type PARAM_DRC_TYPE_LIM

The parametric DRC of type PARAM_DRC_TYPE_LIM (0x1) is a time domain peak limiter design, which is fully specified in Annex G. For audio decoders that operate in the sub-band domain, PARAM_DRC_TYPE_LIM should be only configured for application at the end of the signal path where time domain processing is possible. A block diagram is depicted in Figure AMD1.3.

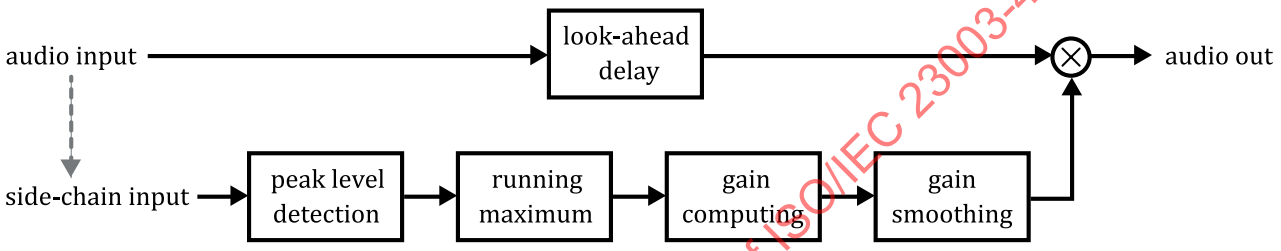

Figure AMD1.3 — Time domain peak limiter design of parametricDrcType==0x1

If not explicitly conveyed in the bitstream, the limiting algorithm will use the following default parameters (see also Table AMD1.78):

— parametricLimThreshold = -1 dBFS

— parametricLimAttackTime = 5 ms

— parametricLimReleaseTime = 50 ms

6.7 Loudness equalization support

The loudEqInstructions() payload provides support for loudness equalization, also known as loudness compensation, which aims to compensate the effect of the playback level on the tonal balance. The loudEqInstructions() repurpose DRC gain sequences to convey the dynamic acoustic level of spectral ranges. Although not described here because it is out of scope, these levels can for instance be used to control dynamic loudness equalization filters.

The loudEqSetId is a unique ID for each loudness equalization (LEQ) set. Each LEQ set declares to which combinations of downmixId and drcSetId it can be applied. Not more than one LEQ set can be declared for a given combination of downmixId and drcSetId. Each LEQ set declares one or more DRC gain sequences it depends on. For each of the gain sequences, the DRC characteristic that was used at the encoder side, the frequency range that should be equalized based on the sequence, a scaling factor, and offset are declared. The time-varying loudness information to control LEQ filters may be derived by applying the inverse DRC characteristic and using the loudness information of the corresponding loudnessInfo() payload. If the inverse characteristic is applied before the interpolation, linear interpolation should be used, i.e. the slope information should be ignored for sequences represented in spline format. The scaling factor and offset are applied in this order after the inverse DRC characteristic in the dB domain. An example LEQ system is discussed in D.2.10.

6.8 Equalization tool

6.8.1 Overview

The equalization (EQ) tool provides EQ filters that are applied to EQ channel groups. The set of EQ filters applied to all EQ channel groups is called "EQ set". EQ can be applied in the time domain or the sub-band domain, for instance, in the QMF domain of an audio decoder. A time-domain EQ filter for a channel group is called a "filter cascade" because it is composed of a cascade of filter blocks, where each block contains one or more filter elements that work in parallel as shown in Figure AMD1.4.

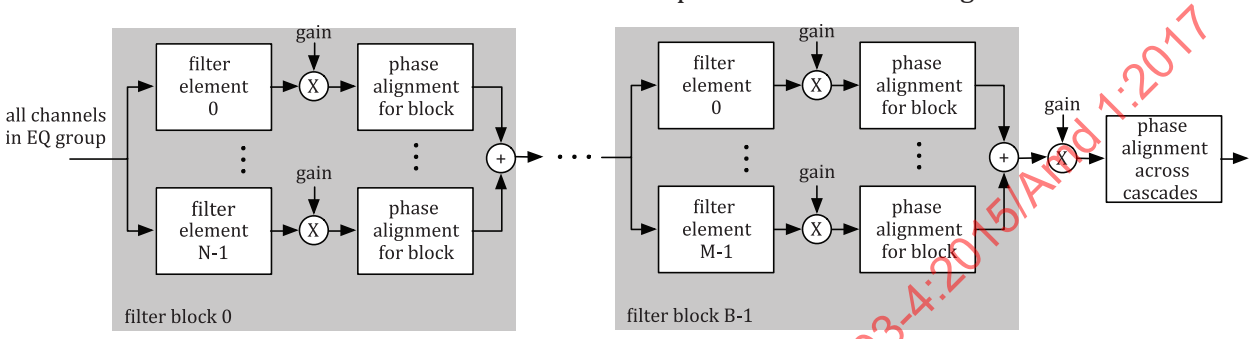

Figure AMD1.4 — Generic EQ filter cascade structure

EQ can be applied before and/or after a downmix. A bitstream field controls whether phase alignment is applied. If the field is set, the EQ filter cascades are phase aligned across EQ channel groups to avoid undesired phase effects. In the sub-band domain, the EQ effect is obtained by applying sub-band gains to channel groups of the audio signal.

If EQ is applied together with DRC, the DRC gain sequences shall be time aligned with the equalized audio signal.

The EQ parameters are conveyed by the payloads, eqCoefficients() and eqInstructions() as described in the following.

6.8.2 EQ payloads

6.8.2.1 Overview

The EQ payloads are transmitted in the uniDrcConfigExtension() payload. They consist of two parts:

— eqCoefficients() including the EQ filter parameters;

— eqInstructions(), each including an EQ set to achieve a certain effect.

The loudness information for EQ processed audio is reflected in the loudnessInfoV1() payload.

6.8.2.2 eqCoefficients() payload

The eqCoefficients() payload in Table AMD1.35 defines all available filter elements and filter blocks. Filter elements can be conveyed in three different formats:

— Pole/zero

— FIR coefficients

— Sub-band gains

The two time-domain formats, pole/zero and FIR coefficients, are mutually exclusive, i.e. only one of them can be used. Sub-band gains can be specified per band or using a spline representation. The different filter elements are indexed. The filter blocks are composed of one ore more filter elements

referred to by an index and an optional filter element gain factor. The available filter blocks can be referenced by the eqInstructions() payload by index.

6.8.2.3 eqInstructions() payload

The eqInstructions() payload in Table AMD1.38 contains metadata to generate a complete EQ filter set for all EQ channel groups. For each channel group, a filter cascade is composed of one or more filter blocks that are defined in the eqCoefficients() payload and a cascade gain factor. Phase alignment flags indicate whether phase alignment is active for two or more channel groups.

The EQ set is externally referred to by an ID specified in the eqSetId field. The downmixId and additionalDownmixIds indicate which downmixes the EQ set can be applied to. The eqApplyToDownmix flag controls whether the EQ is applied before or after a downmix. A second EQ set may be required, depending on the dependsOnEqSet field. The second EQ set shall be located at the opposite side of the downmix block. The EQ set cannot be applied without a second EQ set if the noIndependentEqUse flag is set. One of multiple DRC sets specified by an ID shall be applied. Note that a drcSetId of 0 indicates that applying no DRC is a permitted option.

If the *dependsOnEqSet* field in an eqInstructions() payload has a value of 1, the auxillary information of the payload describes the combined EQ including the dependent one. For the combined EQ, the auxillary information of the dependent eqInstructions() is ignored. The auxiliary information includes (*additional*)*downmixID*, (*additional*)*drcSetId*, and *eqPurpose*.

The eqTransitionDuration indicates the duration of the crossfade between the output of the previous EQ set and the current EQ set when the EQ changes mid-stream. The default value for eqTransitionDuration is 0.05 s.

6.8.2.4 LoudnessInfo payload for EQ

The loudnessInfoV1() payload is identical with the loudnessInfo() payload except for the additional eqSetId field that indicates which EQ set is active. The loudnessInfoV1() payload is transmitted in the loudnessInfoSetExtension() payload.

6.8.3 EQ filter elements

6.8.3.1 Supported filter elements in pole/zero format

A filter element has the following generic pole/zero form in the complex *z*-domain:

$$H_{\text{element,PZ}}(z) = g_{\text{element}} \frac{\prod_{m-1}^{M}\left(1 - c_m z^{-1}\right)}{\prod_{k=1}^{K/2}\left(1 - d_k z^{-1}\right)^2}$$

The gain coefficient g_{element} is a real value. The class of filters that can be used as a filter element has several constraints.

Constraints for all the poles, assuming a pole has a radius, *r*, and angle, θ ($d_k = re^{j\theta}$):

— *K* is even or zero.

— All poles are inside the unit circle ($|r|<1$).

— For each pole, there is a second pole at the same location.

— If a complex pole is present, the conjugate complex pole shall be present as well (if $\theta \neq n\pi : re^{j\theta}, re^{-j\theta}$ are present).

Constraints for all the zeros, assuming a zero has a radius, *r*, and angle, θ ($c_m = re^{j\theta}$):

— *M* is even or zero.

— An even number of real zeros with |r|==1 can be present.

— Each real zero with |r| ≠ 1 is part of a pair of zeros at $r, \frac{1}{r}$.

— Each generic zero is part of a quadruple of zeros at $re^{j\theta}, re^{-j\theta}, \frac{1}{r}e^{j\theta}, \frac{1}{r}e^{-j\theta}$.

The constraints for the poles ensure that the filter is stable and that an all-pass filter with the same phase response can be created. The constraints for the zeros ensure that the corresponding FIR filter has linear phase and a constant integer delay when measured in units of a sample interval. Please note that further restrictions according to Table AMD1.11 apply to filter elements for which an all-pass filter shall be created to achieve phase alignment.

6.8.3.2 Supported filter elements in FIR coefficient format

In general, a filter element in FIR coefficient format can be any symmetrical FIR filter (symmetric or anti-symmetric). Taking advantage of the symmetry, only the first half of the coefficients is transmitted. An FIR filter can only be used, if it is not considered in any phase alignment as described in 6.8.5.1.2. Therefore, it can only be used in filter blocks that have only a single filter element and if the filter block is not part of a filter cascade with phase alignment.

6.8.3.3 Supported filter elements in sub-band gain format

Sub-band gains represent the spectral shape of the EQ filter. The gains are applied to the sub-band signals, typically in the QMF domain of an audio decoder.

6.8.4 EQ set selection

Conceptually, the EQ set selection has lower priority than the DRC set selection. Therefore, the DRC set selection is processed first and if there are still different EQ sets during the final selection step, the EQ set is selected, which is appropriate for the requested EQ purpose. If there is no EQ request from the host, a default EQ is requested. DRCs that are automatically applied, such as for ducking or fading, are not affected by the EQ selection. A mid-stream change of the EQ payload does not trigger the selection process, i.e. the selected EQ set ID continues to be valid but an EQ crossfade may be needed if the corresponding EQ filter parameters have changed.

The selected EQ set may have a dependent EQ set that shall be applied as well.

6.8.5 Application of EQ set

The selected EQ sets are applied to EQ channel groups of the audio signal before and/or after the downmix. Figure AMD1.5 shows an example processing chain with EQ and DRC. If a DRC set is applied at the same location before or after the downmix, the EQ set shall be applied before the DRC set. Please note that the channel grouping for EQ and DRC is independent.

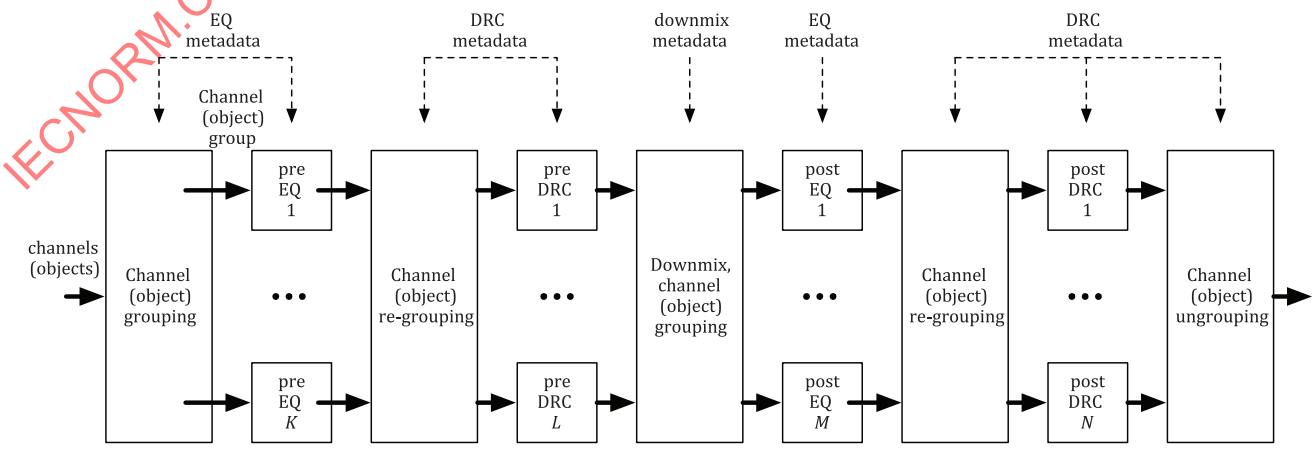

Figure AMD1.5 — Example overview of processing chain with EQ, DRC, and downmix

If the EQ payload changes or a different EQ is selected mid-stream, the outputs of the previous EQ processing and current EQ processing are cross-faded for the specified duration according to the *bsTransitionDuration* field using a linear cross-fade characteristic.

6.8.5.1 Time domain EQ

6.8.5.1.1 Decoding of filter element parameters

6.8.5.1.1.1 Decoding of *z*-domain poles and zeros

Due to the filter element constraints, only a fraction of the actual pole and zero locations needs to be transmitted. The remaining ones are recovered by the decoder according to Table AMD1.11 and Table AMD1.12. Therefore, it is sufficient to transmit zeros and poles in the range of $r = [0,1]$ and $\theta = [0,\pi]$ only.

Table AMD1.11 — Transmitted and decoded pole locations ($n \in \{0,1\}$)

Transmitted pole; associated bitstream count field	Decoded poles
Real pole: $re^{j\theta}$, ($r<1$, $\theta == n\pi$); realPoleCount	Real double pole: $re^{j\theta}$, $re^{j\theta}$
Complex pole : $re^{j\theta}$, ($r<1, \theta \neq n\pi$); complexPoleCount	Double conjugate pole pair: $re^{j\theta}$, $re^{j\theta}$, $re^{-j\theta}$, $re^{-j\theta}$

Table AMD1.12 — Transmitted and decoded zero locations ($n \in \{0,1\}$)

Transmitted zero; associated bitstream count field	Decoded zero(s)
Generic zero: $re^{j\theta}$, (any r, any θ); genericZeroCount	Two conjugate zero pairs: $re^{j\theta}$, $re^{-j\theta}$, $\frac{1}{r}e^{j\theta}$, $\frac{1}{r}e^{-j\theta}$
Real zero (for restricted use)[a]: $re^{j\theta}$, ($r == 1$, $\theta == n\pi$); realZeroRadiusOneCount	Real zero: $re^{j\theta}$
Real zero[b]: $re^{j\theta}$, ($r<1$, $\theta == n\pi$); realZeroCount	Inverse zero pair: $re^{j\theta}$, $\frac{1}{r}e^{j\theta}$

[a] Only permitted if used in filter blocks that contain only a single filter element and if these filter blocks are not subject to cascade phase alignment.

[b] For each real zero with $\theta == 0$, multiply the linear filter element gain factor by -1 to compensate for the phase inversion.

The radius, r, of a pole or zero is decoded according to Table AMD1.95 and by computing the radius from ρ using $r = 1 - \rho$. The ρ values of Table AMD1.95 can be generated using the result for *r_quant_offs* of the pseudo code in Table AMD1.13.

Table AMD1.13 — Pseudo code to generate the zero/pole radius decoding table

```
r_quant_step_min = 0.2;
for (n=0; n<128; n++) {
    tmp[n]=1.0/(0.025*(127-n));
}
for (n=0; n<128; n++) {
    r_quant_step[n] = tmp[n] + r_quant_step_min - tmp[1];
}
sumlevel[0] = r_quant_step[0];
for (n=1; n<128; n++) {
    sumlevel[n] = sumlevel[n-1] + r_quant_step[n];
}
```

Table AMD1.13 *(continued)*

```
for (n=0; n<128; n++) {
    r_quant[n] = 1 - pow(10.0, -0.05*sumlevel[n]);
    r_quant_offs[n] = pow(10.0, -0.05*sumlevel[n]);
}
```

The angle θ of a pole or zero is decoded according to Table AMD1.96 and by computing the angle θ by $\theta = \alpha\pi$. The table can be generated by the pseudo code in Table AMD1.14 using the result for *angle_quant*. For real poles or zeros, the angle is determined by the associated sign field. If the sign is 1, the angle is $\theta = \pi$, otherwise, $\theta = 0$.

Table AMD1.14 — Pseudo code to generate the pole/zero angle decoding table

```
for (n=0; n<12; n++) {          /* 12 steps per octave */
    octave[n] = pow(2.0, -n/12.0);
}
for (k=0; k<11; k++) {
    for (n=0; n<12; n++) {
         tmp[k*12 + n] = octave[n] * pow(2.0, -k);
    }
}
tmp[127] = 0.0;
for (i=0; i<127; i++) {
    angle_quant = tmp[127-i];
}
```

Once all poles and zeros are decoded, a corresponding filter is created. It is recommended that each pair of poles and pair of zeros are grouped and implemented using a second-order section. The remaining zeros are implemented by an FIR filter. In detail, the recommended steps are:

a) The pole that is closest to the unit circle should be paired with the zero that is closest to it in the *z*-plane where the radius of zeros with $r > 1$ is replaced by the inverse radius $1/r$ for the distance calculation.

b) Rule (1) should be repeatedly applied until all the poles have been paired with zeros.

c) The resulting second-order sections should be ordered according to decreasing closeness of poles to the unit circle.

d) The remaining zeros are processed using an FIR filter.

6.8.5.1.1.2 Decoding of FIR coefficients

The FIR coefficients are decoded according to Table AMD1.93. Since only the first half of the coefficients is transmitted, the second half is generated according to Table AMD1.15.

Table AMD1.15 — Pseudo code to generate the FIR filter coefficients

```
for (i=0; i<firFilterOrder/2+1; i++) {
    firCoef[i] = decodeFirCoef(bsFirCoefficient[i]);
}
for (i=0; i<(firFilterOrder+1)/2; i++) {
    if (firSymmetry==1) {
        firCoef[firFilterOrder-i] = - firCoef[i];
    } else {
        firCoef[firFilterOrder-i] = firCoef[i];
    }
```

Table AMD1.15 *(continued)*

```
}
if ((firSymmetry==1) && ((firFilterOrder & 1)==0)) {
    firCoef[firFilterOrder/2] = 0.0;
}
```

6.8.5.1.2 Phase alignment of time-domain filter

All-pass filters are employed to achieve phase alignment. Phase alignment is always necessary for all filter elements in each filter block. Phase alignment shall also be done at the output of filter cascades if the corresponding phase alignment flag is set.

For phase alignment on a filter block level, the phase response of each filter element in the block is matched by an all-pass filter. For filter elements in pole/zero format that have more poles than zeros ($K > M$), a delay of $P = (K - M)/2$ shall be added to the filter element so that phase matching can be achieved as described below. The matching all-pass filter is then inserted after all filter elements except the ones with the matching phase response.

The phase alignment at the filter cascade level is controlled by the phase alignment flags. For phase alignment at this level, the phase response of each cascade shall be matched by a corresponding cascade of filter blocks with all-pass filters that match the phase response of the corresponding filter elements. Each all-pass cascade shall then be inserted after all cascades except the one with the matching phase response.

Identical all-pass filters that appear in all parallel paths at the filter block or cascade filter level shall be eliminated. To minimize delay, the aggregated all-pass delays as indicated by L in all parallel paths shall be reduced by the same amount until at least one path has zero delay. Similarly, the delay added to filter elements with $K > M$ shall be reduced as much as possible.

The all-pass with matching phase response of a filter element is derived from the poles and zeros of a filter element in pole/zero format as follows. In mathematical terms, the transfer function of the all-pass filter with a magnitude response of unity can be written as:

$$H_{\text{allpass,PZ}}(z) = \prod_{k=1}^{k/2}\left(-d_k^*\right)\frac{\prod_{k=1}^{K/2}\left(1 - z^{-1}/d_k^*\right)}{\prod_{k=1}^{K/2}\left(1 - d_k z^{-1}\right)} z^{-L}$$

The phase response of a basic EQ filter is matched by the all-pass filter when:

— Each single pole of the all-pass filter corresponds to a pair of poles of the filter element. Both poles are at the same location.

— The location of the zeros of the all-pass filter are given by the inverse conjugate complex pole location of the all-pass filter.

— The delay of L = (M– K)/2 samples is necessary if the filter element has more zeros than poles (M > K).

If the filter element has FIR coefficient format, phase alignment is not supported for that element.

6.8.5.2 Sub-band domain EQ

6.8.5.2.1 Decoding of sub-band gains

Sub-band gains shall be provided that cover the entire audio spectrum. Depending on the sub-band representation of the rendering system, the sub-band gains may need to be interpolated and resampled

using the applicable sub-band center frequencies. A number of specific sub-band formats are supported, for instance, some common QMF configurations. The sub-band gain values shall be present for each band in ascending (frequency) order or the spectral gain function shall be given by a spline representation as indicated by the *eqSubbandGainRepresentation* field.

If *eqSubbandGainRepresentation* == 0, the sub-band gains are decoded according to Table AMD1.91. They are applied to the sub-bands of the audio codec, usually the QMF domain starting with the first gain for the sub-band with the lowest center frequency.

If *eqSubbandGainRepresentation* == 1, the sub-band EQ gains are represented in a spline format. Table AMD1.16 contains pseudo code for the spline interpolation of the gains. The interpolation is done in the dB domain and the gain values are converted to linear values before they are applied to the sub-band signals. The spline nodes are located on a sparse grid with approximately a third octave frequency resolution. The sub-band EQ gains are calculated by evaluating the spline at the center frequency of each sub-band.

Table AMD1.16 — Pseudo code for decoding and spline interpolation to produce an EQ sub-band gain vector gEqSubband[b] for the b-th sub-band

```
eqNodeCountMax = 33;
eqNodeIndexMax = eqNodeCountMax - 1;
sEqNode[0] = decodeEqSlope(eqSlopeCode[0]);
gEqNode[0] = decodeEqInitialGain(eqGainInitialCode);
eqNodeFreqIndex[0] = 0;
eqNodeFreq[0] = decodeEqNodeFreq(eqNodeFreqIndex[0]);
for (n=1; n<nEqNodes; n++) {
    sEqNode[n] = decodeEqSlope(eqSlopeCode[n]);
    gEqNode[n] = gEqNode[n-1] + decodeEqGainDelta(eqGainDeltaCode[n]);
    fEqDelta[n] = decodeEqFreqDelta(eqFreqDeltaCode[n]);
    eqNodeFreqIndex[n] = eqNodeFreqIndex[n-1] + fEqDelta[n];
    eqNodeFreq[n] = decodeEqNodeFreq(eqNodeFreqIndex[n]);
}
if ((eqNodeFreq[nEqNodes-1] < audioSampleFreq/2.0) &&
                    (eqNodeFreqIndex[nEqNodes-1] < eqNodeIndexMax)) {
    sEqNode[nEqNodes] = 0;
    gEqNode[nEqNodes] = gEqNode[nEqNodes-1];
    fEqDelta[nEqNodes] = eqNodeIndexMax - eqNodeFreqIndex[nEqNodes-1];
    eqNodeFreqIndex[nEqNodes] = eqNodeIndexMax;
    eqNodeFreq[nEqNodes] = decodeEqNodeFreq(eqNodeFreqIndex[nEqNodes]);
    nEqNodes += 1;
}
for (n=0; n<nEqNodes-1; n++) {
    for (b=0; b<eqSubbandGainCount; b++) {
        fSub = max(fCenter[b], eqNodeFreq[0]);
        fSub = min(fSub, eqNodeFreq[nEqNodes-1]);
        if ((fSub >= eqNodeFreq[n]) && (fSub <= eqNodeFreq[n+1])) {
            warpedDeltaFreq = warpFreqDelta (fSub, eqNodeFreqIndex[n]);
            gEqSubbandDb[b] = interpolateEqGain(fEqDelta[n+1], gEqNode[n], gEqNode[n+1],
                                                sEqNode[n], sEqNode[n+1], warpedDeltaFreq);
            gEqSubband[b] = pow(2.0, gEqSubbandDb[b] / 6.0);
        }
    }
}
```

Table AMD1.17 — Specifications of parameters and functions used in Table AMD1.16

Parameter/function	Description
audioSampleFreq	Sample rate of audio signal in Hz.
fCenter[b]	Center frequency of *b*-th sub-band in Hz.

fLo = 20;
fHi = 24000;
eqNodeCountMax = 33;
stepRatio = (log(fHi) / log(fLo) - 1) / (eqNodeCountMax -1);
	Common parameters.

decodeEqNodeFreq(eqNodeFreqIndex) {
return(pow(fLo, 1 + eqNodeFreqIndex * stepRatio));
}	

warpFreqDelta(fSubband, eqNodeFreqIndex) {
return((log(fSubband)/log(nodeFrequency[0]) – 1) /
stepRatio – eqNodeFreqIndex);
}	

interpolateEqGain()	See Table AMD1.18.
decodeEqSlope()	See Table AMD1.98.
decodeEqInitialGain()	See Table AMD1.100.
decodeEqGainDelta()	See Table AMD1.101.
decodeEqFreqDelta()	See Table AMD1.99.

Table AMD1.18 — Interpolation of sub-band EQ gains for one spline segment

```
interpolateEqGain(bandStep, gain0, gain1, slope0, slope1, f)
{
    float k1, k2, a, b, c, d;
    float nodesPerOctaveCount = 3.128;
    float gainLeft = gain0;
    float gainRight = gain1;
    float slopeLeft = slope0 / nodesPerOctaveCount;
    float slopeRight = slope1 / nodesPerOctaveCount;
    float bandStepInv = 1.0 / (float)bandStep;
    float bandStepInv2 = bandStepInv * bandStepInv;
    k1 = (gainRight - gainLeft) * bandStepInv2;
    k2 = slopeRight + slopeLeft;
    a = bandStepInv * (bandStepInv * k2 - 2.0 * k1);
    b = 3.0 * k1 - bandStepInv * (k2 + slopeLeft);
    c = slopeLeft;
    d = gainLeft;
    result = (((a * f + b ) * f + c ) * f ) + d;
    return result;
}
```

If sub-band gains are not available but the EQ set is applied to the sub-band domain, the sub-band gains are computed from the time-domain filter elements at the sub-band center frequencies $\omega = \omega_c$ (*bandIndex*) in radians.

The magnitude response of a single zero is:

$$\left|H_{\text{zero},m}\left(e^{j\omega}\right)\right| = \left[1 + r^2 - 2r\cos\left(\omega - \theta\right)\right]^{0.5}$$

The magnitude response of a single pole is:

$$\left|H_{\text{pole},k}\left(e^{j\omega}\right)\right| = \left[1 + r^2 - 2r\cos\left(\omega - \theta\right)\right]^{-0.5}$$

Hence, the magnitude of a filter element in pole/zero format is determined by:

$$\left|H_{\text{element, PZ}}\left(e^{j\omega}\right)\right| = g_{\text{element}} \prod_{m=1}^{M}\left|H_{\text{zero},m}\left(e^{j\omega}\right)\right| \prod_{k=1}^{K}\left|H_{\text{pole},k}\left(e^{j\omega}\right)\right|$$

For the FIR coefficient format, the magnitude response of the filter element can be computed as shown in Table AMD1.19, assuming that the impulse response is denoted by $h(n)$.

Table AMD1.19 — Magnitude response of FIR filter

FIR type	Order *M*	Sym-metry	Magnitude response $\lvert H_{element}(e^{jw})\rvert$	Coefficients
1	Even	Even	$g_{\text{element}} \sum_{m=0}^{M/2} \alpha(m)\cos(m\omega)$	$a(0) = h\left(\frac{M}{2}\right)$ $\alpha(m) = 2h\left(\frac{M}{2} - m\right),\ m \in \left[1, \frac{M}{2}\right]$
3	Even	Odd	$g_{\text{element}} \sum_{m=1}^{M/2} \alpha(m)\sin(m\omega)$	
2	Odd	Even	$g_{\text{element}} \sum_{m=1}^{(M+1)/2} \beta(m)\cos\left(\left(m - \frac{1}{2}\right)\omega\right)$	$\beta(m) = 2h\left(\frac{(M+1)}{2} - m\right),$ $m \in \left[1, \frac{M+1}{2}\right]$
4	Odd	Odd	$g_{\text{element}} \sum_{m=1}^{(M+1)/2} \beta(m)\sin\left(\left(m - \frac{1}{2}\right)\omega\right)$	

The magnitude response of a filter cascade with B filter blocks where each block has $E(b)$ filter elements is therefore:

$$\left|H_{\text{cascade}}\left(e^{j\omega}\right)\right| = g_{\text{cascade}} \prod_{b=1}^{B}\left[\sum_{e=1}^{E(b)}\left|H_{\text{element},e,b}\left(e^{j\omega}\right)\right|\right]$$

The transmission of time-domain EQ filters only is sufficient to also cover sub-band EQ. However, if only sub-band gains are transmitted without specifying the time-domain EQ filters, Table AMD1.20 should be used to determine the applicable EQ. Please note that time-domain filters shall be specified if the EQ is to be applied in the time-domain.

Table AMD1.20 — Conversion of EQ filter formats

Transmitted EQ formats	Domain where EQ is applied	Conversion
Time domain and sub-band domain	Sub-band	Compute sub-band gains from time-domain filters if EQ sub-band resolution does not match and sub-band gains are not available in spline format.
Time domain	Sub-band	Compute sub-band gains from time-domain filters.
Sub-band	Sub-band	If the EQ gains are not given in spline format, resample using linear interpolation in dB domain, if necessary.
Sub-band	Time-domain	EQ cannot be applied.

6.9 Complexity management

A DRC tool decoder implementation should determine or include a value $L_{C,\mathrm{tot,max}}$ for the absolute total complexity level permitted for processing. Based on that value, the DRC tool does not permit configurations that exceed the complexity limit based on the complexity estimation. Therefore, a configuration can only be applied if the following condition is met:

$$\log_2(C_{\mathrm{DRC,tot}} + C_{\mathrm{EQ,tot}}) \leq L_{C,\mathrm{tot,max}}$$

The definitions of $C_{\mathrm{DRC,tot}}$ and $C_{\mathrm{EQ,tot}}$ are given in the following subclauses. Application standards and audio coding standards that include this specification may require that $L_{C,\mathrm{tot,max}}$ exceeds a given minimum value.

It is recommended that the DRC tool decoder disables all DRC sets with $\log_2(C_{\mathrm{DRC,tot}}) > L_{C,\mathrm{tot,max}}$ and all EQ sets with $\log_2(C_{\mathrm{EQ,tot}}) > L_{C,\mathrm{tot,max}}$ before the DRC selection process. The combined complexity of DRC and EQ can be evaluated after the DRC selection process. If it exceeds the permitted complexity, the EQ should be disabled, unless the *requiresEQ* field of the DRC has a value of 1. If the complexity still exceeds the limit, the primary DRC should be disabled and the DRC selection process should be reiterated.

6.9.1 DRC and downmixing complexity estimation

The total complexity $C_{\mathrm{DRC,tot}}$ for DRC processing and downmixing is estimated in the DRC tool decoder based on the *drcSetComplexityLevel* field value $L_{C,\mathrm{DRC}}$ in the drcInstructionsUniDrcV1() payloads and the downmix matrix, if applied.

$$C_{\mathrm{DRC,tot}} = \frac{f_s}{f_{s,\mathrm{norm}}}\left(\sum_{m=1}^{3} N_{ch,m}\, 2^{L_{C,\mathrm{DRC},m}} + \sum_{i=1}^{N_{\mathrm{Base}}}\sum_{j=1}^{N_{\mathrm{Target}}} D_{i,j}\right)$$

The parameters are defined as:

f_s output audio sample rate in Hz.

$f_{s,\mathrm{norm}}$ constant of 48 000 Hz.

$L_{C,\mathrm{DRC},1}$ represents the complexity level of a primary DRC, if present. Otherwise, it is 0.

$L_{C,\mathrm{DRC},2}$ represents the complexity level of a dependent DRC, if present. Otherwise, it is 0.

$L_{C,\mathrm{DRC},3}$ represents the complexity level of a ducking or fading DRC, if present. Otherwise, it is 0.

$N_{\mathrm{ch},m}$ number of audio channels at the location of DRC_m, if applicable. Otherwise, it is 0.

N_{Base} audio channel count of base layout.

N_{Target} audio channel count after downmix.

$D_{i,j}$ has a value of 0 if no downmix is applied or if the corresponding downmix coefficient has a value of 0; otherwise, it is 1 for time-domain processing or 2 for sub-band processing.

The DRC tool encoder computes the complexity level for each DRC set as follows:

$$L_{C,\mathrm{DRC}} = \max(0,\mathrm{ceil}(L'_{C,\mathrm{DRC}}))$$

with

$$L'_{C,\mathrm{DRC}} = \log_2\left(\frac{1}{I}\left(\sum_{i=1}^{I}\left(w_{\mathrm{Mod}}P_i + w_{IIR}Q_i + w_{Lap}R_i + w_{Shape}S_i\right) + \sum_{k=1}^{K}T_k\right)\right)$$

DRC sets with $L'_{C,\mathrm{DRC}} > 15$ are not permitted. The parameters are defined as:

I audio channel count at the DRC location. If the corresponding drcInstructionsUniDrcV1() payload includes a downmixId of 0×7F, it is the channel count of the base layout.

w_{Mod} weighting factor that considers the complexity of applying the DRC gain to the audio signal. It has a value of 1.0 for time-domain processing or 2.0 for sub-band processing.

P_i has a value of 1 if a DRC set is applied to channel i. Otherwise, it is 0.

w_{IIR} weighting factor of 5.0 to consider the IIR filter complexity.

Q_i number of pairs of a single zero and pole which occur in the Linkwitz-Riley filters for time-domain band splitting including all phase-alignment filters in audio channel i. The value is zero for a single band time-domain DRC or any sub-band domain DRC. For time-domain DRCs with more than four bands, the Linkwitz-Riley filter bank cannot be used. A decoder can support such DRCs by using a QMF analysis and synthesis bank instead. However, in that case, the complexity of the QMF processing is not included in the drcSetComplexityLevel field and Q_i has a value of 0.

w_{Lap} weighting factor of 2.0 for the complexity of the sub-band overlap processing.

R_i number of DRC bands for audio channel i, if a multiband DRC is applied in the sub-band domain and drcBandType==1. Otherwise, the value is 0.

w_{Shape} weighting factor of 6.0 for the complexity of the shaping filter processing.

S_i number of pairs of a single zero and pole that occur in the shaping filter bank for audio channel i. The value is 0 if no shaping filter is applied.

K number of different DRC sequences that are actively applied to the audio signal.

T_k complexity value for computing DRC gain values of each gain sequence k that is applied to the audio signal according to Table AMD1.21.

Table AMD1.21 — Complexity values for computing DRC gains. (*A*: channel count at side chain input, *B*: weighting filter order, C: parametricLimAttackTime, D: default value of C)

Gain sequence generation	DRC domain	*T*
Spline interpolation (*interpolationType* == 0)	Time domain	5.0
Linear interpolation (*interpolationType* == 1)	Time domain	2.5
Spline interpolation (*interpolationType* == 0)	Sub-band	0.0
Linear interpolation (*interpolationType* == 1)	Sub-band	0.0
Parametric DRC (*parametricDrcType*==0)	Time domain	$A(5B + 1) + 3$
Parametric DRC (*parametricDrcType*==0)	Sub-band	$5A$
Parametric limiter (*parametricDrcType*==1)	Time domain	$4.5A + 136\sqrt{C / D}$

The complexity of a dependent DRC set is not included in the result. $L_{C,\mathrm{DRC}}$ is 0 if *bsSequenceIndex*==0. The *drcSetComplexityLevel* value applies to the domain where the DRC is processed by default given a fully bitstream compatible audio decoder.

To ensure that the DRC tool decoder implementation supports a certain complexity level, tests should include edge-case configurations that individually maximize the complexity contribution of each one of the parameters in the complexity estimation formula.

6.9.2 EQ complexity estimation

The total complexity $C_{\mathrm{EQ,tot}}$ for EQ processing is estimated in the decoder based on the *eqSetComplexityLevel* field value $L_{\mathrm{C,EQ}}$ in the eqInstructions() payloads:

$$C_{\mathrm{EQ,tot}} = \frac{f_{\mathrm{s}}}{f_{\mathrm{s,norm}}} \sum_{i=1}^{2} N_{ch,i}\, 2^{L_{C,\mathrm{EQ},i}}$$

The parameters are defined as:

f_{s} output audio sample rate in Hz.

$f_{\mathrm{s,norm}}$ constant of 48 000 Hz.

$L_{C,\mathrm{EQ},1}$ represents the complexity level of a primary EQ set, if present. Otherwise, it is 0.

$L_{C,\mathrm{EQ},2}$ represents the complexity level of a dependent EQ set, if present. Otherwise, it is 0.

$N_{ch,1}$, $N_{ch,2}$ represent the audio channel count at the location before and after the downmix, where the primary and dependent EQ sets are applied, if present. Otherwise, it is 0.

The DRC tool encoder computes the complexity level for each EQ set by:

$$L_{C,\mathrm{EQ}} = \max(0, \mathrm{ceil}(L'_{\mathrm{C,EQ}}))$$

with

$$L'_{C,EQ} = \log_2\left(\frac{1}{I}\sum_{i=1}^{I}\left(w_{Sb}U_i + w_{IIR}V_i + w_{FIR}Z_i\right)\right)$$

EQ sets with $L'_{C,EQ} > 15$ are not permitted. The parameters are defined as:

I audio channel count at the EQ location. If the corresponding eqInstructions() payload includes a downmixID of 0x7F, it is the channel count of the base layout.

w_{Sb} weighting factor of 2.5 that considers the complexity of applying the EQ gain to the sub-band signal.

U_i has a value of 1 if the EQ is applied in the sub-band domain of channel i. Otherwise, it is 0.

w_{IIR} weighting factor of 5.0 that considers the IIR filter complexity.

V_i number of pairs of a single zero and pole that occur in IIR filters including all phase-alignment filters applied to channel i in the time domain, if present. Otherwise, it is 0.

w_{FIR} weighting factor of 0.4 that considers the FIR filter complexity.

Z_i total order of all FIR filters applied to channel i in the time domain, if present. Otherwise, it is 0.

$L'_{C,EQ}$ is 0 if the EQ instructions of both, tdFilterCascadePresent and subbandGainsPresent have a value of 0 and the audio signal is processed in the time-domain. The complexity of a dependent EQ set is not included in the result.

The *eqSetComplexityLevel* value applies to the domain where the EQ set is specified. If it is specified in both domains, it applies to the time domain and the sub-band domain EQ complexity is set to 2.5. If the EQ filters are specified in the time domain but applied in the sub-band domain, the EQ complexity value for the sub-band domain is also set to 2.5.

Note that the EQ can include filter elements consisting of a combination of an IIR and FIR filter. To ensure that the DRC tool decoder implementation supports a certain complexity level, tests should include the time-domain edge-case configurations at the maximum supported complexity for EQ with IIR filters only ($Z_i = 0$) and EQ with FIR filters only ($V_i = 0$).

Page 41, 6.6

Replace Table 27 with:

Table 27 — Measurement system indices for the loudness normalization settings

Index	Measurement system
0	Default system
1	ITU-R BS.1770-4
2	User
3	Expert/Panel
4 *(reserved, not permitted)*	Reserved Measurement System A (RMS_A)
5 *(reserved, not permitted)*	Reserved Measurement System B (RMS_B)
6 *(reserved, not permitted)*	Reserved Measurement System C (RMS_C)
7 *(reserved, not permitted)*	Reserved Measurement System D (RMS_D)
8 *(reserved, not permitted)*	Reserved Measurement System E (RMS_E)

Replace Table 29 with:

Table 29 — Default loudness normalization settings

Entry	Default value	Meaning of default
Measurement method	1	Program Loudness
Measurement system	1	ITU-R BS.1770-4
Pre-processing	1	No pre-processing

Replace Table 30 with:

Table 30 — Matching order for measurement system

Order of selection	Requested measurement system as defined in Table 27							
	BS.1770-4	**User**	**Expert/ Panel**	**RMS_A**	**RMS_B**	**RMS_C**	**RMS_D**	**RMS_E**
	Selected measurement system as defined in Table A.37							
1	BS.1770-4	User	Expert/ Panel	RMS_A	RMS_B	RMS_C	RMS_D	RMS_E
2	RMS_C	RMS_E	RMS_D	RMS_C	RMS_C	RMS_B	Expert/ Panel	User
3	RMS_B	RMS_D	RMS_C	RMS_B	RMS_A	RMS_A	RMS_C	RMS_D
4	RMS_A	Expert/ Panel	RMS_B	BS.1770-4	BS.1770-4	BS.1770-4	RMS_B	Expert/ Panel
5	RMS_D	RMS_C	RMS_A	RMS_D	RMS_D	RMS_D	RMS_A	RMS_C
6	Expert/ Panel	RMS_B	BS.1770-4	Expert/ Panel	Expert/ Panel	Expert/ Panel	BS.1770-4	RMS_B
7	RMS_E	RMS_A	RMS_E	RMS_E	RMS_E	RMS_E	RMS_E	RMS_A
8	User	BS.1770-4	User	User	User	User	User	BS.1770-4

Page 42, 6.6.2

Replace paragraph for step 3 with:

3. If pre-processing is requested, the loudness value needs to be adjusted. The adjustment value ΔL_{pre} the difference between the program loudness based on ITU-R BS.1770-4 with pre-processing and ITU-R BS.1770-4 without pre-processing. If the two loudness values are not available, the adjustment value is set to -2 dB. The value of contentLoudness is equal to the selected loudness value plus adjustment.

Page 46, 7.2

Replace Table 34:

Table 34 — Syntax of uniDrcGain() payload

Syntax	No. of bits	Mnemonic

uniDrcGain()
{
  nDrcGainSequences=sequenceCount;
  /* from drcCoefficientsUniDrc */
  for (s=0; s<nDrcGainSequences; s++) {
  if (gainCodingProfile[s]<3) {
  drcGainSequence();
  }
  }
  uniDrcGainExtPresent;
	1	**bslbf**

if (uniDrcGainExtPresent==1) {
  uniDrcGainExtension();
  }
}		

With:

Table 34 — Syntax of uniDrcGain() payload

Syntax	No. of bits	Mnemonic

uniDrcGain()
{
  nDrcGainSequences=gainSequenceCount;
  /* from drcCoefficientsUniDrc or drcCoefficientsUniDrcV1 */
  for (s=0; s<nDrcGainSequences; s++) {
  if (gainCodingProfile[s]<3) {
  drcGainSequence();
  }
  }
  uniDrcGainExtPresent;
	1	**bslbf**

if (uniDrcGainExtPresent==1) {
  uniDrcGainExtension();
  }
}		

Page 48, 7.3

Add a new table after Table 38:

Table AMD1.22 — Syntax of in-stream loudnessInfoV1() payload

Syntax	No. of bits	Mnemonic
loudnessInfoV1()		
{		
drcSetId;	**6**	**uimsbf**
eqSetId;	**6**	**uimsbf**
downmixId;	**7**	**uimsbf**
samplePeakLevelPresent;	**1**	**bslbf**
if (samplePeakLevelPresent==1) {		
bsSamplePeakLevel;	**12**	**uimsbf**
}		
truePeakLevelPresent;	**1**	**bslbf**
if (truePeakLevelPresent==1) {		
bsTruePeakLevel;	**12**	**uimsbf**
measurementSystem;	**4**	**uimsbf**
reliability;	**2**	**uimsbf**
}		
measurementCount;	**4**	**uimsbf**
for (i=0; i<measurementCount; i++) {		
methodDefinition;	**4**	**uimsbf**
methodValue;	**2..8**	**vlclbf**
measurementSystem;	**4**	**uimsbf**
reliability;	**2**	**uimsbf**
}		
}		

Page 49, 7.3

Replace Table 39:

Table 39 — Syntax of loudnessInfoSetExtension() payload

Syntax	No. of bits	Mnemonic
loudnessInfoSetExtension()		
{		
while (**loudnessInfoSetExtType** != UNIDRCLOUDEXT_TERM) {;	**4**	**uimsbf**
extSizeBits = **bitSizeLen** + 4;	**4**	**uimsbf**
extBitSize = **bitSize** + 1;	**extSizeBits**	**uimsbf**
switch (loudnessInfoSetExtType) {		
default:		
for (i=0; i<extBitSize; i++) {		
otherBit;	**1**	**bslbf**

Table 39 *(continued)*

Syntax	No. of bits	Mnemonic
} } }		

With:

Table 39 — Syntax of loudnessInfoSetExtension() payload

Syntax	No. of bits	Mnemonic

loudnessInfoSetExtension()
{
while (**loudnessInfoSetExtType** != UNIDRCLOUDEXT_TERM) {
	4	**uimsbf**

extSizeBits = **bitSizeLen** + 4;	**4**	**uimsbf**
extBitSize = **bitSize** + 1;	**extSizeBits**	**uimsbf**
switch (loudnessInfoSetExtType) {		
UNIDRCLOUDEXT_EQ:		
loudnessInfoV1AlbumCount;	**6**	**uimsbf**
loudnessInfoV1Count;	**6**	**uimsbf**
for (i=0; i<loudnessInfoV1AlbumCount; i++) {		
loudnessInfoV1(); }		
for (i=0; i<loudnessInfoV1Count; i++) {		
loudnessInfoV1(); }		
break;		
default:		
for (i=0; i<extBitSize; i++) {		
otherBit;	**1**	**bslbf**
} } } }		

Page 50, 7.3

Add a new table after Table 41:

Table AMD1.23 — Syntax of in-stream downmixInstructionsV1() payload

Syntax	No. of bits	Mnemonic
downmixInstructionsV1()		
{		
downmixId;	**7**	**uimsbf**
targetChannelCount;	**7**	**uimsbf**
targetLayout;	**8**	**uimsbf**
downmixCoefficientsPresent;	**1**	**bslbf**
if (downmixCoefficientsPresent==1) {		
bsDownmixOffset;	**4**	**uimsbf**
for (i=0; i<targetChannelCount; i++) {		
for (j=0; j<baseChannelCount; j++) {		
bsDownmixCoefficientV1;	**5**	**bslbf**
}		
}		
}		
}		

Page 50, 7.3

Replace Table 44:

Table 44 — Syntax of drcCoefficientsUniDrc() payload

Syntax	No. of bits	Mnemonic
drcCoefficientsUniDrc()		
{		
drcLocation;	**4**	**uimsbf**
drcFrameSizePresent;	**1**	**bslbf**
if (drcFrameSizePresent==1) {		
bsDrcFrameSize;	**15**	**uimsbf**
}		
sequenceCount;	**6**	**uimsbf**
for (i=0; i<sequenceCount; i++) {		
gainCodingProfile;	**2**	**uimsbf**
gainInterpolationType;	**1**	**uimsbf**
fullFrame;	**1**	**uimsbf**
timeAlignment;	**1**	**uimsbf**
timeDeltaMinPresent;	**1**	**bslbf**
if (timeDeltaMinPresent==1) {		
bsTimeDeltaMin;	**11**	**uimsbf**
}		
if (gainCodingProfile!=3) {		
bandCount;	**4**	**uimsbf**
if (bandCount>1) {		
drcBandType;	**1**	**uimsbf**
}		

Table 44 *(continued)*

Syntax	No. of bits	Mnemonic
for (j=0; j<bandCount; j++) {		
drcCharacteristic;	**7**	**uimsbf**
}		
for (j=1; j<bandCount; j++) {		
if (drcBandType==1) {		
crossoverFreqIndex;	**4**	**uimsbf**
} else {		
startSubBandIndex;	**10**	**uimsbf**
}		
}		
}		
}		
}		

With:

Table 44 — Syntax of in-stream drcCoefficientsUniDrc() payload

Syntax	No. of bits	Mnemonic
drcCoefficientsUniDrc()		
{		
drcLocation;	**4**	**uimsbf**
drcFrameSizePresent;	**1**	**bslbf**
if (drcFrameSizePresent==1) {		
bsDrcFrameSize;	**15**	**uimsbf**
}		
gainSetCount;	**6**	**uimsbf**
k=0;		
for (i=0; i<gainSetCount; i++) {		
gainSetIndex = i;		
gainCodingProfile;	**2**	**uimsbf**
gainInterpolationType;	**1**	**uimsbf**
fullFrame;	**1**	**uimsbf**
timeAlignment;	**1**	**uimsbf**
timeDeltaMinPresent;	**1**	**bslbf**
if (timeDeltaMinPresent==1) {		
bsTimeDeltaMin;	**11**	**uimsbf**
}		
if (gainCodingProfile==3) {		
bandCount=1;		
} else {		
bandCount;	**4**	**uimsbf**
if (bandCount>1) {		
drcBandType;	**1**	**uimsbf**
}		
for (j=0; j<bandCount; j++) {		

Table 44 *(continued)*

Syntax	No. of bits	Mnemonic
drcCharacteristic;	**7**	**uimsbf**
}		
for (j=1; j<bandCount; j++) {		
if (drcBandType==1) {		
crossoverFreqIndex;	**4**	**uimsbf**
} else {		
startSubBandIndex;	**10**	**uimsbf**
}		
}		
}		
k = k + bandCount;		
}		
gainSequenceCount = k;		
}		

Page 50, 7.3

Add a new table after Table 44:

Table AMD1.24 — Syntax of in-stream drcCoefficientsUniDrcV1() payload

Syntax	No. of bits	Mnemonic
drcCoefficientsUniDrcV1()		
{		
drcLocation;	**4**	**uimsbf**
drcFrameSizePresent;	**1**	**bslbf**
if (drcFrameSizePresent==1) {		
bsDrcFrameSize;	**15**	**uimsbf**
}		
drcCharacteristicLeftPresent;	**1**	**bslbf**
if (drcCharacteristicLeftPresent == 1) {		
characteristicLeftCount;	**4**	**uimsbf**
for (k=1; k<=characteristicLeftCount; k++) {		
characteristicLeftIndex = k;		
characteristicFormat;	**1**	**bslbf**
if (characteristicFormat==0) {		
bsGainLeft;	**6**	**uimsbf**
bsIoRatioLeft;	**4**	**uimsbf**
bsExpLeft;	**4**	**uimsbf**
flipSignLeft;	**1**	**bslbf**
} else {		

NOTE 1 The value of this field shall be identical for all DRC gain sets that share one or more DRC gain sequences.

NOTE 2 The values of the index field shall fulfill the condition bsIndex < gainSequenceCount.

Table AMD1.24 *(continued)*

Syntax	No. of bits	Mnemonic
characteristicNodeCount = bsCharNodeCount+1;	2	uimsbf
for (n=1; n<=characteristicNodeCount; n++) {		
bsNodeLevelDelta;	5	uimsbf
bsNodeGain;	8	uimsbf
}		
}		
}		
}		
drcCharacteristicRightPresent;	1	bslbf
if (drcCharacteristicRightPresent == 1) {		
characteristicRightCount ;	4	uimsbf
for (k=1 ;k<=characteristicRightCount; k++) {		
characteristicRightIndex = k;		
characteristicFormat;	1	bslbf
if (characteristicFormat==0) {		
bsGainRight;	6	uimsbf
bsIoRatioRight;	4	uimsbf
bsExpRight;	4	uimsbf
flipSignRight;	1	bslbf
} else {		
characteristicNodeCount = bsCharNodeCount+1;	2	uimsbf
for (n=1; n<=characteristicNodeCount; n++) {		
bsNodeLevelDelta;	5	uimsbf
bsNodeGain;	8	uimsbf
}		
}		
}		
}		
shapeFiltersPresent;	1	bslbf
if (shapeFiltersPresent == 1) {		
shapeFilterCount;	4	uimsbf
for (k=1; k<=shapeFilterCount; k++) {		
shapeFilterIndex = k;		
lfCutFilterPresent;	1	bslbf
if (lfCutFilterPresent == 1) {		
lfCornerFreqIndex;	3	uimsbf
lfFilterStrengthIndex;	2	uimsbf
}		
lfBoostFilterPresent;	1	bslbf

NOTE 1 The value of this field shall be identical for all DRC gain sets that share one or more DRC gain sequences.

NOTE 2 The values of the index field shall fulfill the condition bsIndex < gainSequenceCount.

Table AMD1.24 *(continued)*

Syntax	No. of bits	Mnemonic
if (lfBoostFilterPresent == 1) {		
lfCornerFreqIndex;	**3**	**uimsbf**
lfFilterStrengthIndex;	**2**	**uimsbf**
}		
hfCutFilterPresent;	**1**	**bslbf**
if (hfCutFilterPresent == 1) {		
hfCornerFreqIndex;	**3**	**uimsbf**
hfFilterStrengthIndex;	**2**	**uimsbf**
}		
hfBoostFilterPresent;	**1**	**bslbf**
if (hfBoostFilterPresent == 1) {		
hfCornerFreqIndex;	**3**	**uimsbf**
hfFilterStrengthIndex;	**2**	**uimsbf**
}		
}		
}		
gainSequenceCount;	**6**	**uimsbf**
gainSetCount;	**6**	**uimsbf**
gainSequenceIndex = -1;		
for (i=0; i<gainSetCount; i++) {		
gainSetIndex = i;		
gainCodingProfile; /* Note (1) */	**2**	**uimsbf**
gainInterpolationType; /* Note (1) */	**1**	**uimsbf**
fullFrame; /* Note (1) */	**1**	**uimsbf**
timeAlignment; /* Note (1) */	**1**	**uimsbf**
timeDeltaMinPresent; /* Note (1) */	**1**	**bslbf**
if (timeDeltaMinPresent==1) {		
bsTimeDeltaMin; /* Note (1) */	**11**	**uimsbf**
}		
if (gainCodingProfile==3) {		
gainSequenceIndex += 1;		
} else {		
bandCount;	**4**	**uimsbf**
if (bandCount>1) {		
drcBandType;	**1**	**uimsbf**
}		
for (j=0; j<bandCount; j++) {		
indexPresent;	**1**	**bslbf**
if (indexPresent == 1) {		
gainSequenceIndex = **bsIndex;** /* Note (2) */	**6**	**uimsbf**
} else {		
gainSequenceIndex += 1;		
}		

NOTE 1 The value of this field shall be identical for all DRC gain sets that share one or more DRC gain sequences.

NOTE 2 The values of the index field shall fulfill the condition bsIndex < gainSequenceCount.

Table AMD1.24 *(continued)*

Syntax	No. of bits	Mnemonic
drcCharacteristicPresent;	**1**	**bslbf**
if (drcCharacteristicPresent) {		
drcCharacteristicFormatIsCICP;	**1**	**bslbf**
if (drcCharacteristicFormatIsCICP) {		
drcCharacteristic;	**7**	**uimsbf**
} else {		
drcCharacteristicLeftIndex;	**4**	**uimsbf**
drcCharacteristicRightIndex;	**4**	**uimsbf**
}		
} else {		
drcCharacteristic = 0;		
drcCharacteristicLeftIndex = 0;		
drcCharacteristicRightIndex = 0;		
}		
}		
for (j=1; j<bandCount; j++) {		
if (drcBandType==1) {		
crossoverFreqIndex;	**4**	**uimsbf**
} else {		
startSubBandIndex;	**10**	**uimsbf**
}		
}		
}		
}		
}		

NOTE 1 The value of this field shall be identical for all DRC gain sets that share one or more DRC gain sequences.

NOTE 2 The values of the index field shall fulfill the condition bsIndex < gainSequenceCount.

Page 52, 7.3

Replace Table 45:

Table 45 — Syntax of drcCoefficientsUniDrc() payload for ISO/IEC 14496-12

```
aligned(8) class DRCCoefficientsUniDrc extends FullBox('udc2') {
            // N copies of this box, one of these per DRC_location
   bit(2)  reserved = 0;
   signed int(5) DRC_location;
   bit(1)  drc_frame_size_present;
   if (drc_frame_size_present == 1) {
       bit(1)  reserved = 0;
       unsigned int(15) bs_drc_frame_size;
   }
```

Table 45 *(continued)*

```
    bit(1)  reserved = 0;
    int(1)  delayMode;
    unsigned int(6) sequence_count;
    for (sequence_index=1; sequence_index<=sequence_count; sequence_index++){
            // each entry here in 1:1 correspondence with the
            // gain coefficient sets (for the bands) in the given location
        bit(2)  reserved = 0;
        unsigned int(2) gain_coding_profile;
        unsigned int(1) gain_interpolation_type;
        unsigned int(1) full_frame;
        unsigned int(1) time_alignment;
        bit(1) time_delta_min_present;
        if (time_delta_min_present == 1) {
            bit(5) reserved = 0;
            unsigned int(11) bs_time_delta_min;
        }
        if (gain_coding_profile!=3) {
            bit(3)  reserved = 0;
            unsigned int(4) band_count;  // must be >= 1
            unsigned int(1) drc_band_type;
            for (j = 1; j <= band_count; j++) {
                bit(1) reserved = 0;
                unsigned int(7) DRC_characteristic;
            }
            for (j = 2; j <= band_count; j++){
                if (drc_band_type == 1) {
                   bit(4) reserved = 0;
                   unsigned int(4) crossover_freq_index;
                } else {
                   bit(6) reserved = 0;
                   unsigned int(10) start_sub_band_index;
                }
            }
        }
    }
}
```

With:

Table 45 — Syntax of drcCoefficientsUniDrc() payload for ISO/IEC 14496-12

```
aligned(8) class DRCCoefficientsUniDrc extends FullBox('udc2', version, flags=0) {
                // N copies of this box, one of these per DRC_location
    bit(2)          reserved = 0;
    signed int(5)   DRC_location;
    bit(1)      drc_frame_size_present;
```

Table 45 *(continued)*

```
if (drc_frame_size_present == 1) {
   bit(1)              reserved = 0;
   unsigned int(15) bs_drc_frame_size;
}
if (version >= 1} {
   bit(5) reserved = 0;
   bit(1) drc_characteristic_left_present;
   bit(1) drc_characteristic_right_present;
   bit(1) shape_filters_present;
   if (drc_characteristic_left_present == 1) {
       bit(4) reserved = 0;
       unsigned int(4) characteristic_left_count;
       for (k=1; k<=characteristic_left_count; k++) {
           bit(7)             reserved = 0;
           unsigned int(1)    characteristic_format;
           if (characteristic_format==0) {
              bit(1)             reserved = 0;
              unsigned int(6)     bs_gain_left;
              unsigned int(4)     bs_io_ratio_left;
              unsigned int(4)     bs_exp_left;
              bit(1)      flip_sign_left;
           } else {
              bit(6)             reserved = 0;
              unsigned int(2)    bs_char_node_count;
              for (n=1; n<=bs_char_node_count+1; n++) {
                   bit(3)             reserved = 0;
                   unsigned int(5)    bs_node_level_delta;
                   unsigned int(8)    bs_node_gain;
              }
           }
       }
   }
   if (drc_characteristic_right_present == 1) {
       bit(4)  reserved = 0;
       unsigned int(4) characteristic_right_count;
       for (k=1; k<=characteristic_right_count; k++) {
           bit(7)            reserved = 0;
           unsigned int(1)   characteristic_format;
           if (characteristic_format==0) {
               bit(1)               reserved = 0;
               unsigned int(6)      bs_gain_right;
               unsigned int(4)      bs_io_ratio_right;
               unsigned int(4)      bs_exp_right;
               bit(1)         flip_sign_right;
           } else {
               bit(6)               reserved = 0;
               unsigned int(2)      bs_char_node_count;
```

Table 45 *(continued)*

```
                for (n=1; n<=bs_char_node_count+1; n++) {
                    bit(3)           reserved = 0;
                    unsigned int(5)   bs_node_level_delta;
                    unsigned int(8)   bs_node_gain;
                }
            }
        }
    }
    if (shape_filters_present==1) {
       bit(4)       reserved = 0;
       unsigned int(4)   shape_filter_count;
       for (k=1; k<=shape_filter_count; k++) {
            bit(4)    reserved = 0;
            bit(1)    LF_cut_filter_present;
            bit(1)    LF_boost_filter_present;
            bit(1)    HF_cut_filter_present;
            bit(1)    HF_boost_filter_present;
            if (LF_cut_filter_present) {
                bit(3)      reserved = 0;
                unsigned int(3)   LF_corner_freq_index;
                unsigned int(2)   LF_filter_strength_index;
            }
            if (LF_boost_filter_present) {
                bit(3)      reserved = 0;
                unsigned int(3)   LF_corner_freq_index;
                unsigned int(2)   LF_filter_strength_index;
            }
            if (HF_cut_filter_present) {
                bit(3)      reserved = 0;
                unsigned int(3)   HF_corner_freq_index;
                unsigned int(2)   HF_filter_strength_index;
            }
            if (HF_boost_filter_present) {
                bit(3)      reserved = 0;
                unsigned int(3)   HF_corner_freq_index;
                unsigned int(2)   HF_filter_strength_index;
            }
        }
    }
  }
  bit(1)            reserved = 0;
  unsigned int(1)   delayMode;
  if (version >= 1} {
     bit(2)           reserved = 0;
     unsigned int(6)  gain_sequence_count;
  }
  unsigned int(6)   gain_set_count;
```

Table 45 *(continued)*

```
    for (i=1; i<=gain_set_count; i++) {
       bit(2)   reserved = 0;
       unsigned int(2) gain_coding_profile;
       unsigned int(1) gain_interpolation_type;
       unsigned int(1) full_frame;
       unsigned int(1) time_alignment;
       bit(1) time_delta_min_present;
       if (time_delta_min_present == 1) {
          bit(5)   reserved = 0;
          unsigned int(11) bs_time_delta_min;
       }
       if (gain_coding_profile!=3) {
          bit(3)   reserved = 0;
          unsigned int(4) band_count;  // must be >= 1
          unsigned int(1) drc_band_type;
          for (j = 1; j <= band_count; j++) {
               if (version>=1) {
                   unsigned int(6)    bs_index;
                   bit(1)          drc_characteristic_present
                   bit(1)           drc_characteristic_format_is_CICP
                   if (drc_characteristic_present==1) {
                      if (drc_characteristic_format_is_CICP==1) {
                          bit(1)          reserved;
                          unsigned int(7)    drc_characteristic;
                      } else {
                          unsigned int    (4)drc_characteristic_left_index;
                          unsigned int(4)    drc_characteristic_right_index;
                      }
                   }
               } else {
                   bit(1)            reserved = 0;
                   unsigned int(7)   drc_characteristic;
               }

          for (j = 2; j <= band_count; j++){
               if (drc_band_type == 1) {
                   bit(4)            reserved = 0;
                   unsigned int(4)   crossover_freq_index;
               } else {
                   bit(6)            reserved = 0;
                   unsigned int(10) start_sub_band_index;
               }
          }
       }
    }
}
```

Page 54, 7.3

Replace Table 48:

Table 48 — Syntax of in-stream drcInstructionsUniDrc() payload

Syntax	No. of bits	Mnemonic
drcInstructionsUniDrc()		
{		
drcSetId;	**6**	**uimsbf**
drcLocation;	**4**	**uimsbf**
downmixId;	**7**	**uimsbf**
additionalDownmixIdPresent;	**1**	**bslbf**
if (additionalDownmixIdPresent==1) {		
additionalDownmixIdCount;	**3**	**uimsbf**
for (j=0; j<additionalDownmixIdCount; j++) {		
additionalDownmixId;	**7**	**uimsbf**
}		
} else {		
additionalDownmixIdCount = 0;		
}		
drcSetEffect;	**16**	**bslbf**
if ((drcSetEffect & (3<<10)) == 0) {		
limiterPeakTargetPresent;	**1**	**bslbf**
if (limiterPeakTargetPresent==1) {		
bsLimiterPeakTarget;	**8**	**uimsbf**
}		
}		
drcSetTargetLoudnessPresent;	**1**	**bslbf**
if (drcSetTargetLoudnessPresent==1) {		
bsDrcSetTargetLoudnessValueUpper;	**6**	**uimsbf**
drcSetTargetLoudnessValueLowerPresent;	**1**	**bslbf**
if (drcSetTargetLoudnessValueLowerPresent==1) {		
bsDrcSetTargetLoudnessValueLower;	**6**	**uimsbf**
}		
}		
dependsOnDrcSetPresent;	**1**	**bslbf**
if (dependsOnDrcSetPresent==1) {		
dependsOnDrcSet;	**6**	**uimsbf**
} else {		
noIndependentUse;	**1**	**bslbf**
}		
channelCount = baseChannelCount;		
if ((drcSetEffect & (3<<10)) != 0) {		
for (i=0; i<channelCount; i++) {		
bsSequenceIndex;	**6**	**uimsbf**
duckingScalingPresent;	**1**	**bslbf**
if (duckingScalingPresent==1) {		
bsDuckingScaling;	**4**	**bslbf**

Table 48 *(continued)*

Syntax	No. of bits	Mnemonic
}		
repeatParameters;	**1**	**bslbf**
if (repeatParameters) {		
bsRepeatParametersCount;	**5**	**uimsbf**
i = i + repeatParametersCount;		
}		
}		
} else {		
if(downmixId!=0 && downmixId!=0x7F) {		
channelCount = targetChannelCountFromDownmixId;		
} else if (downmixId==0x7F) {		
channelCount = 1;		
}		
for (i=0; i<channelCount; i++) {		
bsSequenceIndex;	**6**	**uimsbf**
repeatSequenceIndex;	**1**	**bslbf**
if (repeatSequenceIndex) {		
bsRepeatSequenceCount;	**5**	**uimsbf**
i = i + repeatSequenceCount;		
}		
}		
for (i=0; i<nDrcChannelGroups; i++) {		
gainScalingPresent;	**1**	**bslbf**
if (gainScalingPresent==1) {		
bsAttenuationScaling;	**4**	**uimsbf**
bsAmplificationScaling;	**4**	**uimsbf**
}		
gainOffsetPresent;	**1**	**bslbf**
if (gainOffsetPresent==1) {		
bsGainOffset;	**6**	**bslbf**
}		
}		
}		
}		

With:

Table 48 — Syntax of in-stream drcInstructionsUniDrc() payload

Syntax	No. of bits	Mnemonic
drcInstructionsUniDrc()		
{		
drcSetId;	**6**	**uimsbf**
drcLocation;	**4**	**uimsbf**
downmixId;	**7**	**uimsbf**
additionalDownmixIdPresent;	**1**	**bslbf**
if (additionalDownmixIdPresent==1) {		
additionalDownmixIdCount;	**3**	**uimsbf**
for (j=0; j<additionalDownmixIdCount; j++) {		
additionalDownmixId;	**7**	**uimsbf**
}		
} else {		
additionalDownmixIdCount = 0;		
}		
drcSetEffect;	**16**	**bslbf**
if ((drcSetEffect & (3<<10)) == 0) {		
limiterPeakTargetPresent;	**1**	**bslbf**
if (limiterPeakTargetPresent==1) {		
bsLimiterPeakTarget;	**8**	**uimsbf**
}		
}		
drcSetTargetLoudnessPresent;	**1**	**bslbf**
if (drcSetTargetLoudnessPresent==1) {		
bsDrcSetTargetLoudnessValueUpper;	**6**	**uimsbf**
drcSetTargetLoudnessValueLowerPresent;	**1**	**bslbf**
if (drcSetTargetLoudnessValueLowerPresent==1) {		
bsDrcSetTargetLoudnessValueLower;	**6**	**uimsbf**
}		
}		
dependsOnDrcSetPresent;	**1**	**bslbf**
if (dependsOnDrcSetPresent==1) {		
dependsOnDrcSet;	**6**	**uimsbf**
} else {		
noIndependentUse;	**1**	**bslbf**
}		
channelCount = baseChannelCount;		
if ((drcSetEffect & (3<<10)) != 0) {		
for (i=0; i<channelCount; i++) {		
bsGainSetIndex;	**6**	**uimsbf**
duckingScalingPresent;	**1**	**bslbf**
if (duckingScalingPresent==1) {		
bsDuckingScaling;	**4**	**bslbf**

Table 48 *(continued)*

Syntax	No. of bits	Mnemonic
}		
repeatParameters;	**1**	**bslbf**
if (repeatParameters) {		
bsRepeatParametersCount;	**5**	**uimsbf**
i = i + repeatParametersCount;		
}		
}		
} else {		
if(downmixId!=0 && downmixId!=0x7F && additionalDownmixIdCount==0) {		
channelCount = targetChannelCountFromDownmixId;		
}else if(downmixId==0x7F || additionalDownmixIdPresent==1) {		
channelCount = 1;		
}		
for (i=0; i<channelCount; i++) {		
bsGainSetIndex;	**6**	**uimsbf**
repeatGainSetIndex;	**1**	**bslbf**
if (repeatGainSetIndex) {		
bsRepeatGainSetIndexCount;	**5**	**uimsbf**
i = i + repeatGainSetIndexCount;		
}		
}		
for (i=0; i<nDrcChannelGroups; i++) {		
gainScalingPresent;	**1**	**bslbf**
if (gainScalingPresent==1) {		
bsAttenuationScaling;	**4**	**uimsbf**
bsAmplificationScaling;	**4**	**uimsbf**
}		
gainOffsetPresent;	**1**	**bslbf**
if (gainOffsetPresent==1) {		
bsGainOffset;	**6**	**bslbf**
}		
}		
}		
}		

Page 54, 7.3

Add a new table after Table 48:

Table AMD1.25 — Syntax of in-stream drcInstructionsUniDrcV1() payload

Syntax	No. of bits	Mnemonic
drcInstructionsUniDrcV1()		
{		
drcSetId;	**6**	**uimsbf**
drcSetComplexityLevel;	**4**	**uimsbf**
drcLocation;	**4**	**uimsbf**
channelCount = baseChannelCount;		
downmixIdPresent;	**1**	**bslbf**
if (downmixIdPresent==1) {		
downmixId;	**7**	**uimsbf**
drcApplyToDownmix;	**1**	**bslbf**
additionalDownmixIdPresent;	**1**	**bslbf**
if (additionalDownmixIdPresent==1) {		
additionalDownmixIdCount;	**3**	**uimsbf**
for (j=0; j<additionalDownmixIdCount; j++) {		
additionalDownmixId;	**7**	**uimsbf**
}		
} else {		
additionalDownmixIdCount = 0;		
}		
if ((drcApplyToDownmix == 1) && (downmixId != 0) &&		
(downmixId != 0x7F) && (additionalDownmixIdCount==0)) {		
channelCount = targetChannelCountFromDownmix;		
} else if ((drcApplyToDownmix == 1) && ((downmixId == 0x7F) ||		
(additionalDownmixIdCount!=0))) {		
channelCount = 1;		
}		
} else {		
downmixId = 0;		
}		
drcSetEffect;	**16**	**bslbf**
if ((drcSetEffect & (3<<10)) == 0) {		
limiterPeakTargetPresent;	**1**	**bslbf**
if (limiterPeakTargetPresent==1) {		
bsLimiterPeakTarget;	**8**	**uimsbf**
}		
}		
drcSetTargetLoudnessPresent;	**1**	**bslbf**
if (drcSetTargetLoudnessPresent==1) {		
bsDrcSetTargetLoudnessValueUpper;	**6**	**uimsbf**
drcSetTargetLoudnessValueLowerPresent;	**1**	**bslbf**

Table AMD1.25 *(continued)*

Syntax	No. of bits	Mnemonic
if (drcSetTargetLoudnessValueLowerPresent==1) {		
bsDrcSetTargetLoudnessValueLower;	**6**	**uimsbf**
}		
}		
dependsOnDrcSetPresent;	**1**	**bslbf**
if (dependsOnDrcSetPresent==1) {		
dependsOnDrcSet;	**6**	**uimsbf**
} else {		
noIndependentUse;	**1**	**bslbf**
}		
requiresEq;	**1**	**bslbf**
if ((drcSetEffect & (3<<10)) != 0) {		
for (i=0; i<baseChannelCount; i++) {		
bsGainSetIndex;	**6**	**uimsbf**
duckingScalingPresent;	**1**	**bslbf**
if (duckingScalingPresent==1) {		
bsDuckingScaling;	**4**	**bslbf**
}		
repeatParameters;	**1**	**bslbf**
if (repeatParameters==1) {		
bsRepeatParametersCount;	**5**	**uimsbf**
i = i + repeatParametersCount;		
}		
}		
} else {		
for (i=0; i<channelCount; i++) {		
bsGainSetIndex;	**6**	**uimsbf**
repeatGainSetIndex;	**1**	**bslbf**
if (repeatGainSetIndex==1) {		
bsRepeatGainSetIndexCount;	**5**	**uimsbf**
i = i + repeatGainSetIndexCount;		
}		
}		
for (i=0; i<nDrcChannelGroups; i++) {		
for (k=0; k<bandCount; k++) {		
targetCharacteristicLeftPresent;	**1**	**bslbf**
if (targetCharacteristicLeftPresent==1) {		
targetCharacteristicLeftIndex;	**4**	**uimsbf**
}		
targetCharacteristicRightPresent;	**1**	**bslbf**
if (targetCharacteristicRightPresent==1) {		
targetCharacteristicRightIndex;	**4**	**uimsbf**
}		

Table AMD1.25 *(continued)*

Syntax	No. of bits	Mnemonic
gainScalingPresent;	**1**	**bslbf**
if (gainScalingPresent==1) {		
bsAttenuationScaling;	**4**	**uimsbf**
bsAmplificationScaling;	**4**	**uimsbf**
}		
gainOffsetPresent;	**1**	**bslbf**
if (gainOffsetPresent==1) {		
bsGainOffset;	**6**	**bslbf**
}		
}		
if (bandCount==1) {		
shapeFilterPresent;	**1**	**bslbf**
if (shapeFilterPresent==1) {		
shapeFilterIndex;	**4**	**bslbf**
}		
}		
}		
}		

Page 57, 7.3

Replace Table 49:

Table 49 — Syntax of drcInstructionsUniDrc() payload for ISO/IEC 14496-12

```
aligned(8) class DRCInstructionsUniDrc extends FullBox('udi2') {
       // N copies, one for each overall combination DRC that can be applied
    bit(3)    reserved = 0;
    unsigned int(6) DRC_set_ID;    // must be non-zero and unique
    signed int(5) DRC_location;
    unsigned int(7) downmix_ID;    // if 0 - to base
                                   // if non-0, applies after downmix
                                   // if 0x7f, applies before or after downmix
    unsigned int(3) additional_downmix_ID_count;
    for (j=0; j<additional_downmix_ID_count; j++) {
       bit(1)    reserved = 0;
       unsigned int(7) additional_downmix_ID;
    }
    unsigned int(16) DRC_set_effect;
    if ((DRC_set_effect & (3<<10)) == 0) {
       bit(7)    reserved = 0;
       bit(1)    limiter_peak_target_present;
       if (limiter_peak_target_present == 1) {
           unsigned int(8) bs_limiter_peak_target;
```

Table 49 *(continued)*

```
    }
  }
  bit(7)    reserved = 0;
  bit(1)    DRC_set_target_loudness_present;
  if (DRC_set_target_loudness_present==1) {
     bit(4)    reserved = 0;
     unsigned int(6) bs_DRC_set_target_loudness_value_upper;
     unsigned int(6) bs_DRC_set_target_loudness_value_lower;
  }
  bit(1)    reserved = 0;
  unsigned int(6)    depends_on_DRC_set;
     // if non-zero, must match a DRC that must be applied before this one
     // it must be examined to see if that is before/after downmix
  if (depends_on_DRC_set==0) {
     bit(1) no_independent_use;
  } else {
     bit(1) reserved = 0;
  }
  unsigned int(8) channel_count;
     // if downmix_ID==0x7F, must be 1, and applies to all channels
     // else must match channel count of the signal
  for (i=1; i<=channel_count; i++){
     unsigned int (8) channel_group_index;
         // 1-based channel_group_index for channel
  }
  for (i=1; i<=channel_group_count; i++){
     bit(2) reserved = 0;
        unsigned int(6) bs_sequence_index;
           // if 0, then this channel_group/object is not processed
           // else 1-based index into the DRC sequence array for this location
  }
  if ((DRC_set_effect & (3<<10)) != 0) {
     for (i=1; i<=channel_group_count; i++) {
         bit(7) reserved = 0;
         bit(1) ducking_scaling_present;
         if (ducking_scaling_present==1) {
            bit(4) reserved = 0;
            bit(4) bs_ducking_scaling;
         }
     }
  }
  else {
     for (i=1; i<=channel_group_count; i++) {
         bit(7) reserved = 0;
         bit(1) gain_scaling_present;
         if (gain_scaling_present==1) {
```

Table 49 *(continued)*

```
                unsigned int(4) bs_attenuation_scaling;
                unsigned int(4) bs_amplification_scaling;
            }
            bit(7) reserved = 0;
            bit(1) gain_offset_present;
            if (gain_offset_present==1) {
                bit(2) reserved = 0;
                bit(6) bs_gain_offset;
            }
        }
    }
}
```

With:

Table 49 — Syntax of drcInstructionsUniDrc() payload for ISO/IEC 14496-12

```
aligned(8) class DRCInstructionsUniDrc extends FullBox('udi2', version, flags=1) {
       // if version == 0, N copies, one for each overall combination DRC
    if (version>=1) {
       bit(2)    reserved = 0;
       unsigned int(6) DRC_instructions_count;
    } else {
       unsigned int DRC_instructions_count = 1;
    }
    for (a=1; a<=DRC_instructions_count; a++) {
       if (version==0) {
           bit(3) reserved = 0;
           unsigned int(6) DRC_set_ID;    // must be non-zero and unique
           signed int(5) DRC_location;          unsigned int(7) downmix_ID;
           unsigned int(7) downmix_ID;  // if 0 - to base
                                        // if non-0, applies after downmix
                                        // if 0x7f, applies before or after downmix
           unsigned int(3) additional_downmix_ID_count;
           for (j=1; j<=additional_downmix_ID_count; j++) {
                bit(1)    reserved = 0;
                unsigned int(7) additional_downmix_ID;
           }
       } else {
           unsigned int(6) DRC_set_ID;   // must be non-zero and unique
           unsigned int(4) DRC_complexity_level;
           signed int(5) DRC_location;
           unsigned int(1) downmix_ID_present;
           if (downmix_ID_present==1) {
                bit(5) reserved = 0;
                unsigned int(7) downmix_ID;
                unsigned int(1) DRC_apply_to_downmix;
                unsigned int(3) additional_downmix_ID_count;
                for (j=1; j<=additional_downmix_ID_count; j++) {
```

Table 49 *(continued)*

```
                bit(1) reserved = 0;
                unsigned int(7) additional_downmix_ID;
            }
        }
    }
    unsigned int(16) DRC_set_effect;
    if ((DRC_set_effect & (3<<10)) == 0) {
        bit(7) reserved = 0;
        bit(1) limiter_peak_target_present;
        if (limiter_peak_target_present == 1) {
            unsigned int(8) bs_limiter_peak_target;
        }
    }
    bit(7) reserved = 0;
    bit(1) DRC_set_target_loudness_present;
    if (DRC_set_target_loudness_present==1) {
       bit(4) reserved = 0;
       unsigned int(6) bs_DRC_set_target_loudness_value_upper;
       unsigned int(6) bs_DRC_set_target_loudness_value_lower;
    }
    if (version>=1) {
       bit(1) requiresEq;
    } else {
       bit(1) reserved = 0;
    }
    unsigned int(6) depends_on_DRC_set;
       // if non-zero, must match a DRC that must be applied before this one
       // it must be examined to see if that is before/after downmix
    if (depends_on_DRC_set==0) {
       bit(1) no_independent_use;
    } else {
       bit(1) reserved = 0;
    }
    unsigned int(8) channel_count;
       // if downmix_ID==0x7F, must be 1, and applies to all channels
       // else must match channel count of the signal
    for (i=1; i<=channel_count; i++){
       unsigned int (8) channel_group_index;
          // 1-based channel_group_index for channel
    }
    for (i=1; i<=channel_group_count; i++){
       bit(2) reserved = 0;
       unsigned int(6) bs_gain_set_index;
          // if 0, then this channel_group/object is not processed
          // else 1-based index into the DRC gain set array for this location       }
    if ((DRC_set_effect & (3<<10)) != 0) {
       for (i=1; i<=channel_group_count; i++) {
          bit(7) reserved = 0;
          bit(1) ducking_scaling_present;
          if (ducking_scaling_present==1) {
```

Table 49 *(continued)*

```
                bit(4) reserved = 0;
                bit(4) bs_ducking_scaling;
            }
        }
    }
    else {
        for (i=1; i<=channel_group_count; i++) {
            if (version>=1) {
                bit(4)             reserved = 0;
                unsigned int(4)    band_count;
                for (k=1; k<=band_count; k++) {
                    bit(4) reserved = 0;
                    bit(1) target_characteristic_left_present;
                    bit(1) target_characteristic_right_present;
                    bit(1) gain_scaling_present;
                    bit(1) gain_offset_present;
                    if (target_characteristic_left_present ==1) {
                        bit(4)            reserved = 0;
                        unsigned int(4)   target_characteristic_left_index;
                    }
                    if (target_characteristic_right_present ==1) {
                        bit(4)            reserved = 0;
                        unsigned int(4)   target_characteristic_right_index;
                    }
                    if (gain_scaling_present==1) {
                        unsigned int(4) bs_attenuation_scaling;
                        unsigned int(4) bs_amplification_scaling;
                    }
                    if (gain_offset_present==1) {
                        bit(2)            reserved = 0;
                        unsigned int(6)   bs_gain_offset;
                    }
                }
                if (band_count==1) {
                    bit(1) shape_filter_present;
                    if (shape_filter_present) {
                        bit(3)            reserved = 0;
                        unsigned int(4)   shape_filter_index;
                    } else {
                        bit(7)            reserved = 0;
                    }
                }
            } else {
                bit(7)    reserved = 0;
                bit(1) gain_scaling_present;
                if (gain_scaling_present==1) {
                    unsigned int(4) bs_attenuation_scaling;
                    unsigned int(4) bs_amplification_scaling;
                }
                bit(7) reserved = 0;
```

Table 49 *(continued)*

```
                bit(1) gain_offset_present;
                if (gain_offset_present==1) {
                    bit(2) reserved = 0;
                    bit(6) bs_gain_offset;
                }
            }
        }
    }
}
```

Page 58, 7.3

Replace Table 50:

Table 50 — Syntax of uniDrcConfigExtension() payload

Syntax	No. of bits	Mnemonic

uniDrcConfigExtension()
{
  while (**uniDrcConfigExtType** != UNIDRCCONFEXT_TERM) {
  extSizeBits = **bitSizeLen** + 4;
	4	**uimsbf**

	4	**uimsbf**
extBitSize = **bitSize** + 1;	**extSizeBits**	**uimsbf**

switch (uniDrcConfigExtType) {
/* add future extensions here */
default:
for (i=0; i<extBitSize; i++) {
otherBit;
	1	**bslbf**

} } } }		

With:

Table 50 — Syntax of uniDrcConfigExtension() payload

Syntax	No. of bits	Mnemonic
uniDrcConfigExtension()		
{		
while (**uniDrcConfigExtType** != UNIDRCCONFEXT_TERM) {	**4**	**uimsbf**
extSizeBits = **bitSizeLen** + 4;	**4**	**uimsbf**
extBitSize = **bitSize** + 1;	**extSizeBits**	**uimsbf**
switch (uniDrcConfigExtType) {		
case UNIDRCCONFEXT_PARAM_DRC:		
drcCoefficientsParametricDrc();		
parametricDrcInstructionsCount;	**4**	**uimsbf**
for (i=0; i<parametricDrcInstructionsCount; i++) {		
parametricDrcInstructions ();		
}		
case UNIDRCCONFEXT_V1:		
downmixInstructionsV1Present;	**1**	**bslbf**
if (downmixInstructionsV1Present==1) {		
downmixInstructionsV1Count;	**7**	**uimsbf**
for (i=0; i<downmixInstructionsV1Count; i++) {		
downmixInstructionsV1();		
}		
}		
drcCoeffsAndInstructionsUniDrcV1Present;	**1**	**bslbf**
if (drcCoeffsAndInstructionsUniDrcV1Present==1) {		
drcCoefficientsUniDrcV1Count;	**3**	**uimsbf**
for (i=0; i<drcCoefficientsUniDrcV1Count; i++) {		
drcCoefficientsUniDrcV1();		
}		
drcInstructionsUniDrcV1Count;	**6**	**uimsbf**
for (i=0; i<drcInstructionsUniDrcV1Count; i++) {		
drcInstructionsUniDrcV1();		
}		
}		
loudEqInstructionsPresent;	**1**	**bslbf**
if (loudEqInstructionsPresent==1) {		
loudEqInstructionsCount;	**4**	**uimsbf**
for (i=0; i<loudEqInstructionsCount; i++) {		
loudEqInstructions();		
}		
}		
eqPresent;	**1**	**bslbf**

Table 50 *(continued)*

Syntax	No. of bits	Mnemonic
if (eqPresent==1) {		
eqCoefficients();		
eqInstructionsCount;	**4**	**uimsbf**
for (i=0; i<eqInstructionsCount; i++) {		
eqInstructions();		
}		
}		
break;		
/* add future extensions here */		
default:		
for (i=0; i<extBitSize; i++) {		
otherBit;	**1**	**bslbf**
}		
}		
}		
}		

Page 58, 7.3

Add a new table after Table 50:

Table AMD1.26 — Syntax of uniDrcConfigExtension() payload for ISO/IEC 14496-12

```
aligned(8) class UniDrcConfigExtension extends Box('udex') {
    // the ordering of the following boxes must be maintained
    // we permit one loudness EQ instruction box:
    LoudEQInstructions();
    // we permit one EQ coefficients box:
    EQCoefficients();
    // we permit one EQ instruction box:
    EQInstructions();
    // we permit one parametric DRC coefficients box:
    DRCCoefficientsParametricDrc();
    // we permit one parametric DRC instruction box:
    ParametricDrcInstructions ();
    Box ();  // further boxes as needed
}
```

Page 59, 7.4

Replace Table 51:

Table 51 — Syntax of drcGainSequence() payload

Syntax	No. of bits	Mnemonic

drcGainSequence()
{
  for(b=0; b<nDrcBands[s]; b++) {
  drcGainCodingMode[s][b];
	1	**uimsbf**

if (drcGainCodingMode[s][b] == 0) { **gainInitialCode**[s][b];	**1..11**	**vlclbf**

nNodes[s][b] = 1;
}
else {
k=0;
do {
k++;
}
while (**endMarker**[s][b][k-1] != 1);
	1	**uimsbf**

nNodes[s][b] = k;
if (gainInterpolationType == 0) {
for (k=0; k<nNodes[s][b]; k++) {
slopeCode[s][b][k++];
	1..9	**vlclbf**

}
}
if (fullFrame == 0) {
frameEndFlag[s][b];
	1	**uimsbf**

}
else {
frameEndFlag[s][b] = 1;
}
for (k=0; k<nNodes[s][b]-1; k++) {
timeDeltaCode[s][b][k];
	2..Z+2	**vlclbf**

}
if (frameEndFlag[s][b] == 0) {
k=nNodes[s][b]-1;
timeDeltaCode[s][b][k];
	2..Z+2	**vlclbf**

} **gainInitialCode**[s][b];	**1..11**	**vlclbf**
for (k=1; k<nNodes[s][b]; k++) { **gainDeltaCode**[s][b][k-1];	**1..11**	**vlclbf**
} } } }		

With:

Table 51 — Syntax of drcGainSequence() payload

Syntax	No. of bits	Mnemonic
drcGainSequence()		
{		
drcGainCodingMode;	**1**	**uimsbf**
if (drcGainCodingMode == 0) {		
gainInitialCode;	**1..11**	**vlclbf**
nNodes = 1;		
}		
else {		
k=0;		
do {		
k++;		
}		
while (**endMarker**[k-1] != 1);	**1**	**uimsbf**
nNodes = k;		
if (gainInterpolationType == 0) {		
for (k=0; k<nNodes; k++) {		
slopeCode[k];	**1..9**	**vlclbf**
}		
}		
if (fullFrame == 0) {		
frameEndFlag;	**1**	**uimsbf**
}		
else {		
frameEndFlag = 1;		
}		
for (k=0; k<nNodes-1; k++) {		
timeDeltaCode[k];	**2..Z+2**	**vlclbf**
}		
if (frameEndFlag == 0) {		
k=nNodes-1;		
timeDeltaCode[k];	**2..Z+2**	**vlclbf**
}		
gainInitialCode;	**1..11**	**vlclbf**
for (k=1; k<nNodes; k++) {		
gainDeltaCode[k-1];	**1..11**	**vlclbf**
}		
}		
}		

Page 59, 7.4

Add new subclauses after 7.4:

7.5 Syntax of parametric DRC tool

Table AMD1.27 — Syntax of drcCoefficientsParametricDrc() payload

Syntax	No. of bits	Mnemonic
drcCoefficientsParametricDrc()		
{		
drcLocation;	**4**	**uimsbf**
parametricDrcFrameSizeFormat;	**1**	**bslbf**
if (parametricDrcFrameSizeFormat==0) {		
bsParametricDrcFrameSize;	**4**	**uimsbf**
} else {		
bsDrcFrameSize;	**15**	**uimsbf**
}		
parametricDrcDelayMaxPresent;	**1**	**bslbf**
if (parametricDrcDelayMaxPresent==1) {		
bsParametricDrcDelayMax;	**8**	**uimsbf**
}		
resetParametricDrc;	**1**	**bslbf**
parametricDrcGainSetCount;	**6**	**uimsbf**
for (i=0; i<parametricDrcGainSetCount; i++) {		
parametricDrcId;	**4**	**uimsbf**
sideChainConfigType;	**3**	**bslbf**
if (sideChainConfigType==1) {		
downmixId;	**7**	**uimsbf**
levelEstimChannelWeightFormat;	**1**	**bslbf**
for (j=0; j<channelCountFromDownmixId; j++) {		
if (levelEstimChannelWeightFormat==0) {		
levelEstimChannelWeight[j] = **addChannel;**	**1**	**bslbf**
} else {		
levelEstimChannelWeight[j] = **bsChannelWeight;**	**4**	**uimsbf**
}		
}		
}		
drcInputLoudnessPresent;	**1**	**bslbf**
if (drcInputLoudnessPresent==1) {		
bsDrcInputLoudness;	**8**	**uimsbf**
} else {		
/* provided by loudnessInfoSet() */		
}		
}		
}		

Table AMD1.28 — Syntax of drcCoefficientsParametricDrc() payload for ISO/IEC 14496-12

```
aligned(8) class DRCCoefficientsParametricDrc extends FullBox('pdc1' version=0,
                                                              flags=0) {
    bit(1)          reserved = 0;
    signed int(5)   drc_location;
    bit(1)          parametric_drc_frame_size_format;
    bit(1)          reset_parametric_drc;
    if (parametric_drc_frame_size_format==0) {
       bit(4)             reserved = 0;
       unsigned int(4)    bs_parametric_drc_frame_size;
    } else {
       bit(1)             reserved = 0;
       unsigned int(15)   bs_drc_frame_size;
    }
    bit(1)          reserved = 0;
    bit(1)          parametric_drc_delay_max_present;
    if (parametric_drc_delay_max_present==1){
       unsigned int (8) parametric_drc_delay_max;
    }
    unsigned int(6)     gain_set_count; // parametric_drc_gain_set_count
    for (gain_set_index=1; gain_set_index <= gain_set_count; gain_set_index++) {
       unsigned int(4)    parametric_drc_ID;
       bit(1)             drc_input_loudness_present;
       bit(3)             side_chain_config_type;
       if (drc_input_loudness_present==1) {
           unsigned int(8) bs_drc_input_loudness;
       } else {
           // provided by LoudnessBox
       }
       if (side_chain_config_type==1) {
           bit(1)             reserved = 0;
           unsigned int(7)    downmix_ID;
           bit(1)             reserved = 0;
           unsigned int(7)    channel_count_from_downmix_ID; // must match
           for (j=1; j <= channel_count_from_downmix_ID; j++) {
               unsigned int(4)    bs_channel_weight;
           }
       }
    }
}
```


Table AMD1.29 — Syntax of parametricDrcInstructions() payload

Syntax	No. of bits	Mnemonic
parametricDrcInstructions ()		
{		
parametricDrcId;	**4**	**uimsbf**
parametricDrcLookAheadPresent;	**1**	**bslbf**
If (parametricDrcLookAheadPresent==1) {		

Table AMD1.29 *(continued)*

Syntax	No. of bits	Mnemonic
bsParametricDrcLookAhead;	**7**	**uimsbf**
}		
parametricDrcPresetIdPresent;	**1**	**bslbf**
if (parametricDrcPresetIdPresent==1) {		
parametricDrcPresetId;	**7**	**uimsbf**
} else {		
parametricDrcType;	**3**	**uimsbf**
if (parametricDrcType == PARAM_DRC_TYPE_FF) {		
parametricDrcTypeFeedForward();		
} else if (parametricDrcType == PARAM_DRC_TYPE_LIM) {		
parametricDrcTypeLimiter();		
} else {		
lenSizeBits = **bitSizeLen** + 4;	**3**	**uimsbf**
lenBitSize = **bitSize** + 1;	**lenSizeBits**	**uimsbf**
switch (parametricDrcType) {		
/* add future extensions here */		
default:		
for (i=0; i<lenBitSize; i++) {		
otherBit;	**1**	**bslbf**
}		
break;		
}		
}		
}		
}		

Table AMD1.30 — Syntax of parametricDrcInstructions() payload for ISO/IEC 14496-12

```
aligned(8) class ParametricDrcInstructions extends FullBox('pdi1' version=0,
                                                        flags=0) {
    bit(4)              reserved = 0;
    unsigned int(4)     parametric_DRC_instructions_count;
    for (a=1; a<=parametric_DRC_instructions_count; a++) {
        bit(2)          reserved = 0;
        unsigned in(4) parametric_drc_ID;
        bit(1)          parametric_drc_look_ahead_present;
        bit(1)          parametric_drc_preset_ID_present;
        if (parametric_drc_look_ahead_present==1) {
            bit(1)  reserved = 0;
            unsigned int(7) bs_parametric_drc_look_ahead;
        }
        if (parametric_drc_preset_ID_present==1) {
            bit(1)           reserved = 0;
            unsigned int(7)  parametric_drc_preset_ID;
        } else {
```

Table AMD1.30 *(continued)*

```
bit(5)           reserved = 0;
unsigned int(3)  parametric_drc_type;
if (parametric_drc_type == 0) { /* PARAM_DRC_TYPE_FF */
    bit(3)            reserved = 0;
    unsigned int(2)   level_estim_K_weighting_type;
    bit(1)            level_estim_integration_time_present;
    unsigned int(1)   drc_curve_definition_type;
    bit(1)            drc_gain_smooth_parameters_present;
    if (level_estim_integration_time_present == 1) {
        bit(2)            reserved = 0;
        unsigned int(6)   bs_level_estim_integration_time;
    }
    if (drc_curve_definition_type==0) {
        bit(1)            reserved = 0;
        unsigned int(7)   DRC_characteristic;
    } else {
        bit(5)            reserved = 0;
        unsigned int(3)   bs_node_count; // node_count=bs_node_count+2
        for (j = 1; j <= node_count; j++) {
            if (j==1) {
                bit(2)            reserved = 0;
                unsigned int(6)   bs_node_level_initial;
            } else {
                bit(3)            reserved = 0;
                unsigned int(5)   bs_node_level_delta;
            }
                bit(2)            reserved = 0;
                unsigned int(6)   bs_node_gain;
            }
        }
        if(drc_gain_smooth_parameters_present==0) {
            // default or matching to drcCharacteristic if available
        } else {
            bit(6)            reserved = 0;
            bit(1)            gain_smooth_time_fast_present;
            bit(1)            gain_smooth_hold_off_count_present;
            unsigned int(8)           bs_gain_smooth_attack_time_slow;
            unsigned int(8)           bs_gain_smooth_release_time_slow;
            if(gain_smooth_time_fast_present==1) {
               unsigned int(8)   bs_gain_smooth_attack_time_fast;
               unsigned int(8)   bs_gain_smooth_release_time_fast;
               bit(7)            reserved = 0;
               bit(1)            gain_smooth_threshold_present;
               if(gain_smooth_threshold_present==1){
                  bit(3)            reserved = 0;
                  unsigned int(5)   bs_gain_smooth_attack_threshold;
                  bit(3)            reserved = 0;
                  unsigned int(5)   bs_gain_smooth_release_threshold;
               }
            }
```

Table AMD1.30 *(continued)*

```
                        if(gain_smooth_hold_off_count_present==1) {
                            bit(1)              reserved = 0;
                            unsigned int(7)     bs_gain_smooth_hold_off;
                        }
                    }
                } else if (parametric_drc_type == 1) { /* PARAM_DRC_TYPE_LIM */) {
                    bit(6)    reserved = 0;
                    bit(1)    parametric_lim_threshold_present;
                    bit(1)    parametric_lim_release_time_present;
                    if (parametric_lim_threshold_present==1) {
                        unsigned int(8)    bs_parametric_lim_threshold;
                    }
                    parametric_lim_attack_time = parametric_drc_look_ahead;
                    if (parametric_lim_release_time_present==1) {
                        unsigned int(8)    bs_parametric_lim_release_time;
                    }
                } else {
                    // disable parametric DRC
                }
            }
        }
    }
```

Table AMD1.31 — Syntax of parametricDrcTypeFeedForward() payload

Syntax	No. of bits	Mnemonic
parametricDrcTypeFeedForward()		
{		
levelEstimKWeightingType;	**2**	**uimsbf**
levelEstimIntegrationTimePresent;	**1**	**bslbf**
if (levelEstimationIntegrationTimePresent==1) {		
bsLevelEstimIntegrationTime;	**6**	**uimsbf**
}		
drcCurveDefinitionType;	**1**	**bslbf**
if (drcCurveDefintionType==0) {		
drcCharacteristic; /* according to CICP */	**7**	**uimsbf**
} else {		
bsNodeCount;	**3**	**uimsbf**
for(j=0; j<nodeCount; j++) {		
if (j==0) {		
bsNodeLevelInitial;	**6**	**uimsbf**
} else {		
bsNodeLevelDelta;	**5**	**uimsbf**
}		
bsNodeGain;	**6**	**uimsbf**
}		
}		

Table AMD1.31 *(continued)*

Syntax	No. of bits	Mnemonic
`drcGainSmoothParametersPresent;`	1	bslbf
`if (drcGainSmoothParametersPresent==0) {`		
`/* default or matching to drcCharacteristic if available */`		
`} else {`		
`bsGainSmoothAttackTimeSlow;`	8	uimsbf
`bsGainSmoothReleaseTimeSlow;`	8	uimsbf
`gainSmoothTimeFastPresent;`	1	bslbf
`if (gainSmoothTimeFastPresent==1) {`		
`bsGainSmoothAttackTimeFast;`	8	uimsbf
`bsGainSmoothReleaseTimeFast;`	8	uimsbf
`gainSmoothTresholdPresent;`	1	bslbf
`if (gainSmoothTresholdPresent==1) {`		
`bsGainSmoothAttackThreshold;`	5	uimsbf
`bsGainSmoothReleaseThreshold;`	5	uimsbf
`}`		
`}`		
`gainSmoothHoldOffCountPresent;`	1	bslbf
`if (gainSmoothHoldOffCountPresent==1) {`		
`bsGainSmoothHoldOff;`	7	uimsbf
`}`		
`}`		
`}`		

Table AMD1.32 — Syntax of parametricDrcTypeLimiter() payload

Syntax	No. of bits	Mnemonic
`parametricDrcTypeLimiter()`		
`{`		
`parametricLimThresholdPresent;`	1	bslbf
`if (parametricLimThresholdPresent==1) {`		
`bsParametricLimThreshold;`	8	uimsbf
`}`		
`parametricLimAttackTime = parametricDrcLookAhead;`		
`parametricLimReleaseTimePresent;`	1	bslbf
`if (parametricLimReleaseTimePresent==1) {`		
`bsParametricLimReleaseTime;`	8	uimsbf
`}`		
`}`		

7.6 Syntax of equalization tools

Table AMD1.33 — Syntax of in-stream loudEqInstructions() payload

Syntax	No. of bits	Mnemonic
loudEqInstructions()		
{		
loudEqSetId;	**4**	**uimsbf**
drcLocation;	**4**	**uimsbf**
downmixIdPresent;	**1**	**bslbf**
if (downmixIdPresent==1) {		
downmixId;	**7**	**uimsbf**
additionalDownmixIdPresent;	**1**	**bslbf**
if (additionalDdownmixIdPresent==1) {		
additionalDownmixIdCount;	**7**	**uimsbf**
for (i=0; i<additionalDownmixIdCount; i++) {		
additionalDownmixId;	**7**	**uimsbf**
}		
} else {		
additionalDownmixIdCount = 0;		
}		
} else {		
downmixId = 0;		
}		
drcSetIdPresent;	**1**	**bslbf**
if (drcSetIdPresent==1) {		
drcSetId;	**6**	**uimsbf**
additionalDrcSetIdPresent;	**1**	**bslbf**
if (additionalDrcSetIdPresent ==1) {		
additionalDrcSetIdCount;	**6**	**uimsbf**
for (i=0; i<additionalDrcSetIdCount; i++) {		
additionalDrcSetId;	**6**	**uimsbf**
}		
} else {		
additionalDrcSetIdCount = 0;		
}		
} else {		
drcSetId = 0;		
}		
eqSetIdPresent;	**1**	**bslbf**
if (eqSetIdPresent==1) {		
eqSetId;	**6**	**uimsbf**
additionalEqSetIdPresent;	**1**	**bslbf**
if (additionalEqSetIdPresent ==1) {		
additionalEqSetIdCount;	**6**	**uimsbf**
for (i=0; i<additionalEqSetIdCount; i++) {		
additionalEqSetId;	**6**	**uimsbf**
}		

Table AMD1.33 *(continued)*

Syntax	No. of bits	Mnemonic
} else {		
additionalEqSetIdCount = 0;		
}		
} else {		
eqSetId = 0;		
}		
loudnessAfterDrc;	**1**	**bslbf**
loudnessAfterEq;	**1**	**bslbf**
loudEqGainSequenceCount;	**6**	**uimsbf**
for (i=0; i<loudEqGainSequenceCount; i++) {		
gainSequenceIndex;	**6**	**uimsbf**
drcCharacteristicFormatIsCICP;	**1**	**bslbf**
if (drcCharacteristicFormatIsCICP==1) {		
drcCharacteristic;	**7**	**uimsbf**
} else {		
drcCharacteristicLeftIndex;	**4**	**uimsbf**
drcCharacteristicRightIndex;	**4**	**uimsbf**
}		
frequencyRangeIndex;	**6**	**uimsbf**
bsLoudEqScaling;	**3**	**uimsbf**
bsLoudEqOffset;	**5**	**uimsbf**
}		
}		

Table AMD1.34 — Syntax of loudEqInstructions() payload for ISO/IEC 14496-12

```
aligned(8) class LoudEqInstructions extends FullBox('leqi', version=0, flags=0) {

    bit(4)        reserved = 0;
    unsigned int (4)    loud_EQ_instructions_count;

    for (a=1; a<=loud_EQ_instructions_count; a++) {
        bit(2)        reserved = 0;
        unsigned int(4)    loud_EQ_set_ID;
        signed int(5)  drc_location;
        bit(1)        downmix_ID_present;
        bit(1)        DRC_set_ID_present;
        bit(1)        EQ_set_ID_present;
        bit(1)        loudness_after_EQ;
        bit(1)        loudness_after_DRC;
        if (downmix_ID_present==1) {
           unsigned int(7)     downmix_ID;
           bit(1)        additional_downmix_ID_present;
           if (additional_downmix_ID_present==1) {
               bit(1)          reserved = 0;
```

Table AMD1.34 *(continued)*

```
            unsigned int(7)    additional_downmix_ID_count;
            for (i=1; i<=additional_downmix_ID_count; i++) {
                bit(1)         reserved = 0;
                unsigned int(7)    additional_downmix_ID;
            }
        }
    }
    if (DRC_set_ID_present ==1) {
        bit(1)         reserved = 0;
        unsigned int(6)    DRC_set_ID;
        bit(1)         additional_DRC_set_ID_present;
        if (additional_DRC_set_ID_present==1) {
            bit(2)         reserved = 0;
            unsigned int(6)    additional_DRC_set_ID_count;
            for (i=1; i<=additional_DRC_set_ID_count; i++) {
                bit(2)         reserved = 0;
                unsigned int(6)    additional_DRC_set_ID;
            }
        }
    }
    if (EQ_set_ID_present ==1) {
        bit(1)         reserved = 0;
        unsigned int(6)    EQ_set_ID;
        bit(1)         additional_EQ_set_ID_present;
        if (additional_EQ_set_ID_present==1) {
            bit(2)         reserved = 0;
            unsigned int(6)    additional_EQ_set_ID_count;
            for (i=1; i<=additional_EQ_set_ID_count; i++) {
                bit(2)         reserved = 0;
                unsigned int(6)    additional_EQ_set_ID;
            }
        }
    }
    bit(2)       reserved = 0;
    unsigned int(6)    loud_EQ_gain_sequence_count;
    for (i=1; i<=loud_EQ_gain_sequence_count; i++) {
        bit(1)       reserved = 0;
        unsigned int(6)    gain_sequence_index;
        bit(1)       DRC_characteristic_format_is_CICP;
        if (DRC_characteristic_format_is_CICP==1) {
            bit(1)       reserved = 0;
            unsigned int(7)    DRC_characteristic;
        } else {
            unsigned int(4)    DRC_characteristic_left_index;
            unsigned int(4)    DRC_characteristic_right_index;
        }
        bit(2)        reserved = 0;
        unsigned int(6)    frequency_range_index;
```

Table AMD1.34 *(continued)*

```
            unsigned int(3)    bs_loud_eq_scaling;
            unsigned int(5)    bs_loud_eq_offset;
        }
    }
}
```

Table AMD1.35 — Syntax of in-stream eqCoefficients() payload

Syntax	No. of bits	Mnemonic
eqCoefficients()		
{		
eqDelayMaxPresent;	**1**	**bslbf**
if (eqDelayMaxPresent==1) {		
bsEqDelayMax;	**8**	**uimsbf**
}		
uniqueFilterBlockCount;	**6**	**uimsbf**
for (j=0; j<uniqueFilterBlockCount; j++) {		
filterBlockIndex = j;		
filterElementCount;	**6**	**uimsbf**
for (k=0; k<filterElementCount; k++) {		
filterElementIndex;	**6**	**uimsbf**
filterElementGainPresent;	**1**	**bslbf**
if (filterElementGainPresent == 1) {		
bsFilterElementGain;	**10**	**uimsbf**
}		
}		
}		
uniqueTdFilterElementCount;	**6**	**uimsbf**
for (k=0; k<uniqueTdFilterElementCount; k++) {		
eqFilterFormat;	**1**	**bslbf**
if (eqFilterFormat == 0) { /* pole / zero */		
bsRealZeroRadiusOneCount;	**3**	**uimsbf**
realZeroCount;	**6**	**uimsbf**
genericZeroCount;	**6**	**uimsbf**
realPoleCount;	**4**	**uimsbf**
complexPoleCount;	**4**	**uimsbf**
realZeroRadiusOneCount = 2* bsRealZeroRadiusOneCount;		
for (m=0; m<realZeroRadiusOneCount; m++) {		
zeroSign;	**1**	**uimsbf**
}		
for (m=0; m<realZeroCount; m++) {		
bsZeroRadius;	**7**	**uimsbf**
zeroSign;	**1**	**uimsbf**
}		
for (m=0; m<genericZeroCount; m++) {		
bsZeroRadius;	**7**	**uimsbf**

Table AMD1.35 *(continued)*

Syntax	No. of bits	Mnemonic
bsZeroAngle;	**7**	**uimsbf**
}		
for (m=0; m<realPoleCount; m++) {		
bsPoleRadius;	**7**	**uimsbf**
poleSign;	**1**	**uimsbf**
}		
for (m=0; m<complexPoleCount; m++) {		
bsPoleRadius;	**7**	**uimsbf**
bsPoleAngle;	**7**	**uimsbf**
}		
} else { /* FIR coefficients */		
firFilterOrder;	**7**	**uimsbf**
firSymmetry;	**1**	**bslbf**
for (m=0; m<firFilterOrder/2+1; m++) {		
bsFirCoefficient;	**11**	**uimsbf**
}		
}		
}		
uniqueEqSubbandGainsCount;	**6**	**uimsbf**
if (uniqueEqSubbandGainsCount > 0) {		
eqSubbandGainRepresentation;	**1**	**bslbf**
eqSubbandGainFormat;	**4**	**uimsbf**
switch (eqSubbandGainFormat) {		
case GAINFORMAT_QMF32:		
eqSubbandGainCount = 32;		
break;		
case GAINFORMAT_QMFHYBRID39		
eqSubbandGainCount = 39;		
break;		
case GAINFORMAT_QMF64:		
eqSubbandGainCount = 64;		
break;		
case GAINFORMAT_QMFHYBRID71:		
eqSubbandGainCount = 71		
break;		
case GAINFORMAT_QMF128:		
eqSubbandGainCount = 128		
break;		
case GAINFORMAT_QMFHYBRID135:		
eqSubbandGainCount = 135		
break;		
case GAINFORMAT_UNIFORM:		
case default:		
eqSubbandGainCount = **bsEqGainCount** + 1;	**8**	**uimsbf**
break;		

Table AMD1.35 *(continued)*

Syntax	No. of bits	Mnemonic

}
for (k=0; k< uniqueEqSubbandGainsCount; k++) {
subbandGainsIndex = k;
if (eqSubbandGainRepresentation == 1) {
eqSubbandGainSpline();
} else {
for (m=0; m<eqSubbandGainCount; m++) {
bsEqSubbandGain;
}
}
}
}
}
	9	**uimsbf**

Table AMD1.36 — Syntax of in-stream eqSubbandGainSpline() payload

Syntax	No. of bits	Mnemonic

eqSubbandGainSpline()
{
nEqNodes = **bsEqNodeCount** + 2;
for (k=0; k<nEqNodes; k++) {
eqSlopeCode[k];
}
for (k=1; k<nEqNodes; k++) {
eqFreqDeltaCode[k];
}
eqGainInitialCode;
for (k=1; k<nEqNodes; k++) {
eqGainDeltaCode[k];
}
}
	5
	1..5
	4
	5..7
	5
		uimsbf
		vlclbf
		uimsbf
		vlclbf
		uimsbf

Table AMD1.37 — Syntax of eqCoefficients() payload for ISO/IEC 14496-12

```
aligned(8) class EQCoefficients extends FullBox('ueqc', version=0, flags=0) {
    bit(1)  reserved = 0;
    bit(1)  eq_delay_max_present;
    if (eq_delay_max_present==1){
       bit(8) bs_eq_delay_max;
    }
    unsigned int(6) unique_filter_block_count;
    for (j=1; j<=unique_filter_block_count; j++){
        bit(2)  reserved = 0;
        unsigned int(6) filter_element_count;
```

Table AMD1.37 *(continued)*

```
        for (k=1; k<=filter_element_count; k++){
            bit(1)              reserved = 0;
            unsigned int(6)     filter_element_index;
            bit(1)              bs_filter_element_gain_present;
            if (bs_filter_element_gain_present == 1) {
                bit(6)              reserved = 0;
                unsigned int(10)    bs_filter_element_gain;
            }
        }
    }
    bit(2)  reserved = 0;
    unsigned int(6) unique_td_filter_element_count;
    for (j=1; j<=unique_td_filter_element_count; j++){
        bit(7) reserved = 0;
        bit(1) EQ_filter_format;
        if (EQ_filter_format == 0) {
            bit(1)    reserved = 0;
            unsigned int(3) bs_real_zero_radius_one_count;
            unsigned int(6) real_zero_count;
            unsigned int(6) complex_zero_count;
            unsigned int(4) real_pole_count;
            unsigned int(4) complex_pole_count;
            for (m=1; m<=bs_real_zero_radius_one_count; m++) {
                unsigned int(1) zero_sign;
            }
            bit(bs_real_zero_radius_one_count mod 8) reserved = 0; // byte fill
            for (m=1; m<=real_zero_count; m++) {
                unsigned int(7) bs_zero_radius;
                unsigned int(1) zero_sign;
            }
            for (m=1; m<=complex_zero_count; m++) {
                bit(2)  reserved = 0;
                unsigned int(7) bs_zero_radius;
                unsigned int(7) bs_zero_angle;
            }
            for (m=1; m<=real_pole_count; m++) {
                unsigned int(7) bs_pole_radius;
                unsigned int(1) zero_sign;
            }
            for (m=1; m<=complex_pole_count; m++) {
                bit(2)  reserved = 0;
                unsigned int(7) bs_pole_radius;
                unsigned int(7) bs_pole_angle;
            }
        } else {  // FIR coefficients
            unsigned int(7) FIR_filter_order;
            bit(1) FIR_symmetry;
            bit((11*(FIR_filter_order/2+1)) mod 8) reserved = 0; // byte fill
            for (m=1; m<=FIR_filter_order/2+1; m++) {
```

Table AMD1.37 *(continued)*

```
                unsigned int(11) bs_FIR_coefficients;
            }
        }
    }
    bit(2)  reserved = 0;
    unsigned int(6) unique_EQ_subband_gains_count;
    if (unique_EQ_subband_gains_count > 0) {
        bit(3)  reserved = 0;
        bit(1)  EQ_subband_gain_repesentation;
        unsigned int(4) EQ_subband_gain_format;
        if (EQ_subband_gain_format== GAINFORMAT_UNIFORM) {
            bit(2) reserved = 0;
            unsigned int(6) bs_EQ_gain_count;
        }
        for (k=1; k<=unique_subband_gains_count; k++) {
            if (EQ_subband_gain_repesentation==1) {
                unsigned int(5) bs_EQ_node_count;
                unsigned int n_EQ_nodes = bs_EQ_node_count + 2;
                unsigned int bitsize = 5;
                for (k=1; k<=n_EQ_nodes; k++) {
                    unsigned int(1…5) EQ_slope_code;
                    bitsize = bitsize + sizeof(EQ_slope_code);
                }
                bit(bitsize mod 8) reserved = 0;  // fill to byte boundary
                for (k=2; k<=n_EQ_nodes; k++) {
                    unsigned int(4) EQ_freq_delta_code;
                }
                bit((4 * n_EQ_nodes) mod 8) reserved = 0;  // fill to byte boundary
                unsigned int(5...7) EQ_gain_initial_code;
                bit(8 - sizeof(EQ_gain_initial_code)) reserved = 0; // byte fill
                for (k=2; k<=n_EQ_nodes; k++) {
                    unsigned int(5) EQ_gain_delta_code;
                }
                bit((5 * n_EQ_nodes) mod 8) reserved = 0;  // byte fill
            } else {              // EQ_subband_gain_count derived as in Table 54
                for (m=1; m<=EQ_subband_gain_count; m++) {
                    unsigned int(9) bs_EQ_subband_gain}
                }
                bit((9*EQ_subband_gain_count) mod 8) reserved = 0; // byte fill
            }
        }
    }
}
```

Table AMD1.38 — Syntax of in-stream eqInstructions() payload

Syntax	No. of bits	Mnemonic
eqInstructions()		
{		
eqSetId;	**6**	**uimsbf**
eqSetComplexityLevel;	**4**	**uimsbf**
channelCount = baseChannelCount;		
downmixIdPresent;	**1**	**bslbf**
if (downmixIdPresent==1) {		
downmixId;	**7**	**uimsbf**
eqApplyToDownmix;	**1**	**bslbf**
additionalDownmixIdPresent;	**1**	**bslbf**
if (additionalDdownmixIdPresent==1) {		
additionalDownmixIdCount;	**7**	**uimsbf**
for (i=0; i<additionalDownmixIdCount; i++) {		
additionalDownmixId;	**7**	**uimsbf**
}		
} else {		
additionalDownmixIdCount = 0;		
}		
if ((eqApplyToDownmix == 1) && (downmixId != 0) &&		
(downmixId != 0x7F) && (additionalDownmixIdCount==0)) {		
channelCount = targetChannelCountFromDownmix;		
} else if (downmixId == 0x7F) {		
channelCount = 1;		
}		
} else {		
downmixId = 0;		
}		
drcSetId;	**6**	**uimsbf**
additionalDrcSetIdPresent;	**1**	**bslbf**
if (additionalDrcSetIdPresent ==1) {		
additionalDrcSetIdCount;	**6**	**uimsbf**
for (i=0; i<additionalDrcSetIdCount; i++) {		
additionalDrcSetId;	**6**	**uimsbf**
}		
} else {		
additionalDrcSetIdCount = 0;		
}		
eqSetPurpose;	**16**	**bslbf**
dependsOnEqSetPresent;	**1**	**bslbf**
if (dependsOnEqSetPresent ==1) {		
dependsOnEqSet;	**6**	**uimsbf**
} else {		
noIndependentEqUse;	**1**	**bslbf**

Table AMD1.38 *(continued)*

Syntax	No. of bits	Mnemonic
}		
eqChannelGroupCount = 0;		
for (i=0; i<channelCount; i++) {		
newGroup = 1;		
eqChannelGroupForChannel[i] = eqChannelGroup;	**7**	**uimsbf**
for (k=0; k<i; k++) {		
if (eqChannelGroup == eqChannelGroupForChannel[k]) {		
newGroup = 0;		
}		
}		
if (newGroup == 1) {		
eqChannelGroupCount += 1;		
}		
}		
tdFilterCascadePresent;	**1**	**bslbf**
if (tdFilterCascadePresent == 1) {		
for (i=0; i<eqChannelGroupCount; i++) {		
eqCascadeGainPresent;	**1**	**bslbf**
if (eqCascadeGainPresent==1) {		
bsEqCascadeGain;	**10**	**uimsbf**
}		
filterBlockCount;	**4**	**uimsbf**
for (k=0; k<filterBlockCount; k++) {		
filterBlockIndex;	**7**	**uimsbf**
}		
}		
eqPhaseAlignmentPresent;	**1**	**bslbf**
for (i=0; i<eqChannelGroupCount; i++) {		
for (k=i+1; k<eqChannelGroupCount; k++) {		
if (eqPhaseAlignmentPresent==1) {		
eqPhaseAlignment(i,k) = bsEqPhaseAlignment;	**1**	**bslbf**
} else {		
eqPhaseAlignment(i,k) = 1;		
}		
}		
}		
}		
subbandGainsPresent;	**1**	**bslbf**
if (subbandGainsPresent == 1) {		
for (i=0; i<eqChannelGroupCount; i++) {		
subbandGainsIndex;	**6**	**uimsbf**
}		
}		

Table AMD1.38 *(continued)*

Syntax	No. of bits	Mnemonic
eqTransitionDurationPresent;	**1**	**bslbf**
if (eqTransitionDurationPresent==1) {		
bsEqTransitionDuration;	**5**	**uimsbf**
}		
}		

Table AMD1.39 — Syntax of eqInstructions() payload for ISO/IEC 14496-12

```
aligned(8) class EQInstructions extends FullBox('ueqi', version=0, flags=0) {
    bit(4)       reserved = 0;
    unsigned int (4)   EQ_instructions_count;
    for (a=1; a<=EQ_instructions_count; a++) {
        bit(1)  reserved = 0;
        unsigned int(6) EQ_set_ID;
        bit(1)  downmix_ID_present;
        if (downmix_ID_present==1) {
           bit(7)  reserved = 0;
           unsigned int(7) downmix_ID;
           bit(1)  EQ_apply_to_downmix;
           bit(1)  additional_downmix_ID_present;
           if (additional_downmix_ID_present==1) {
               bit(1) reserved = 0;
               unsigned int(7) additional_downmix_ID_count;
               for (i=1; i<=additional_downmix_ID_count; i++) {
                   bit(1)  reserved = 0;
                   unsigned int(7) additional_downmix_ID;
               }
           }
        }
        bit(1)  reserved = 0;
        unsigned int(6) drc_set_ID;
        bit(1)  additional_drc_set_ID_present;
        if (additional_drc_set_ID_present==1) {
            bit(1)  reserved = 0;
            unsigned int(7) additional_drc_set_ID_count;
            for (i=1; i<=additional_drc_set_ID_count; i++) {
                bit(2)  reserved = 0;
                unsigned int(6) additional_drc_set_ID;
            }
        }
        bit(3)  reserved = 0;
        bit(4)  EQ_complexity_level;
        bit(16)  EQ_set_purpose;
        bit(1) depends_on_EQ_set_present;
        if (depends_on_EQ_set_present==1) {
            bit(2)  reserved = 0;
            unsigned int(6) depends_on_EQ_set;
```

Table AMD1.39 *(continued)*

```
        } else {
            bit(7) reserved = 0;
            bit(1) no_independent_EQ_use;
        }
        bit(7) reserved = 0;
        unsigned int(7) channelCount;
        unsigned int(7) EQ_channel_group_count;
        bit(1) td_filter_cascade_present;
        bit(1) subband_gains_present;
        bit(1) EQ_transition_duration_present;
        if (td_filter_cascade_present == 1) {
            for (i=1; i<=EQ_channel_group_count; i++) {
                bit(3) reserved = 0;
                bit(1) EQ_cascade_gain_present;
                unsigned int(4) filter_block_count;
                if (EQ_cascade_gain_present == 1) {
                    bit(6) reserved = 0;
                    unsigned int(10) bs_EQ_cascade_gain;
                }
                for (k=1; k<=filter_block_count; k++) {
                    bit(1) reserved = 0;
                    unsigned int(7) filter_block_index;
                }
            }
            bit(7) reserved = 0;
            bit(1) EQ_phase_alignment_present;
            if (EQ_phase_alignment_present == 1) {
                unsigned int bitsize = 0;
                for (i=1; i<=EQ_channel_group_count; i++) {
                    for (k=i+1; k<=EQ_channel_group_count; k++) {
                        bit(1) bs_EQ_phase_alignment;
                        bitsize++;
                    }
                }
                bit(bitsize mod 8) reserved = 0;  // fill to byte boundary
            }
        }
        if (subband_gains_present == 1) {
            for (i=1; i<=EQ_channel_group_count; i++) {
                bit(2) reserved = 0;
                unsigned int(6) subband_gains_index;
            }
        }
        if (EQ_transition_duration_present == 1) {
            bit(3)   reserved = 0;
            unsigned int(5) bs_EQ_transition_duration;
        }
    }
}
```

Page 64, A.6.1

Replace Table A.10 with:

Table A.10 — Coding of top level fields of uniDrcConfig() and loudnessInfoSet()

Field label	Encoding	Mnemonic	Decoded value	Description
bsSampleRate	μ 18 bits	uimsbf	$f_S = \mu + 1000$	Audio sample rate in [Hz]
downmixInstructionsCount, downmixInstructionsV1Count	μ 7 bits	uimsbf	$N_D = \mu$	Number of downmixInstructions() and downmixInstructionsV1() blocks, respectively
drcCoefficientsBasicCount, drcCoefficientsUniDrcCount	μ 3 bits	uimsbf	$N_C = \mu$	Number of drcCoefficients blocks
drcInstructionsBasicCount, drcInstructionsUniDrcCount	μ 4, 6 bits	uimsbf	$N_I = \mu$	Number of drcInstructions blocks
loudnessInfoAlbumCount	μ 6 bits	uimsbf	$N_{LA} = \mu$	Number of loudnessInfo() blocks for albums
loudnessInfoCount	μ 6 bits	uimsbf	$N_L = \mu$	Number of loudnessInfo() blocks

Page 64, A.6.2

Replace Table A.11:

Table A.11 — loudnessInfoSet extension types

Symbol	Value of loudnessInfoSetExtType	Purpose
UNIDRCLOUDEXT_TERM	0×0	Termination tag
(reserved)	*(All remaining values)*	For future use

With:

Table A.11 — loudnessInfoSet extension types

Symbol	Value of loudnessInfoSetExtType	Purpose
UNIDRCLOUDEXT_TERM	0×0	Termination tag
UNIDRCLOUDEXT_EQ	0×1	Extension for equalization
(reserved)	*(All remaining values)*	For future use

Page 65, A.6.3

Replace Table A.12:

Table A.12 — UniDrc configuration extension types

Symbol	Value of uniDrcConfigExtType	Purpose
UNIDRCCONFEXT_TERM	0×0	Termination tag
(reserved)	*(All remaining values)*	For future use

With:

Table A.12 — UniDrc configuration extension types

Symbol	Value of uniDrcConfigExtType	Purpose
UNIDRCCONFEXT_TERM	0×0	Termination tag
UNIDRCCONFEXT_PARAM_DRC	0×1	Parametric DRC
UNIDRCCONFEXT_V1	0×2	Efficient multi-band DRC coding, dynamic EQ, loudness EQ
(reserved)	*(All remaining values)*	For future use

Page 65, A.6.4

Replace Table A.13:

Table A.13 — Coding of metadata that appears in multiple logical blocks

Metadata field	Description
drcLocation	See 6.1.2.3
drcSetId	A unique number for each drcInstructions block. 0×0 reserved.
dependsOnDrcSet	Same encoding as *drcSetId*.
downmixId	A unique number for each downmixInstructions() block. 0×00 and 0×7F are reserved. A value of 0×7F in drcInstructions indicates that the DRC set can be applied to any downmix or to the original channel layout. In this case, the DRC set shall contain a single DRC gain sequence that is applied to all channels. A *downmixId* of zero indicates that the DRC set is applied to the base layout. See also ISO/IEC 14496-12.

With:

Table A.13 — Coding of metadata that appears in multiple logical blocks

Metadata field	Description
drcLocation	See 6.1.2.3.
drcSetId	A unique number for each drcInstructions block. 0x0 and 0×3F are reserved. A value of 0×3F in loudnessInfo() indicates that the loudness information can be applied to any DRC including no DRC.
dependsOnDrcSet	Same encoding as *drcSetId*.
downmixId	A unique number for each downmixInstructions() and downmixInstructionsV1() block. 0×00 and 0×7F are reserved. A value of 0×7F in drcInstructions indicates that it is permitted to apply the DRC set to the base layout or any downmix. In this case, the DRC set shall contain a single DRC gain sequence that is applied to all channels. If the downmixId is not present or if it has a value of zero, the DRC set is applied to the base layout. See also ISO/IEC 14496-12. If the V0 version of drcInstructionsUniDrc() includes a value of 0×7F and a downmix is active, the DRC set shall be applied to the downmix. Note that for the V1 version, drcInstructionsUniDrcV1(), the *drcApplyToDownmix* flag indicates whether the DRC set shall be applied to the downmix or to the base layout.

Page 65, A.6.6

Replace Table A.15:

Table A.15 — Coding of metadata in downmixInstructions()

Metadata field	Description
targetLayout	See ChannelConfiguration in ISO/IEC 23001-8 (CICP).
bsDownmixCoefficient	See ISO/IEC 14496-12 (ISO base media file format).

With:

Table A.15 — Coding of metadata in downmixInstructions() and downmixInstructionsV1()

Metadata field	Description
targetLayout	See ChannelConfiguration in ISO/IEC 23001-8 (CICP).
bsDownmixOffset	See ISO/IEC 14496-12:2015/Amd 1:2017 (ISO base media file format).
bsDownmixCoefficient	See ISO/IEC 14496-12:2015 (ISO base media file format).
bsDownmixCoefficientV1	See ISO/IEC 14496-12:2015/Amd 1:2017 (ISO base media file format).

Page 66, A.6.7

Replace Table A.16:

Table A.16 — Coding of metadata in drcCoefficientsBasic() and drcCoefficientsUniDrc()

Metadata field	Description
gainCodingProfile	See Table A.18.
gainInterpolationType	See Table A.19.
fullFrame	A value of 1 signals that the last node is always at the end of the frame. In low-delay mode, it shall be 1.
timeAlignment	A bit field that indicates whether the gain sample is aligned with the end (0) or the center (1) of the *deltaTmin* interval.
delayMode	A bit field that indicates whether the received gains are applied immediately or with a delay of one frame (see Table A.20).
bsTimeDeltaMin	If present, this field indicates the custom time resolution of a DRC sequence which overrides the default *deltaTmin* value. The values are encoded according to Table A.21.
drcCharacteristic	A value of 0 means that the DRC characteristic is undefined for a DRC sequence. Values > 11 are reserved. See ISO/IEC 23001-8 (CICP).
drcBandType	A bit field, which signals if the “crossoverFreqIndex-syntax” (1) or the “startSubBandIndex-syntax” (0) should be used. If multi-band DRC gains are applied in the time-domain by using the multi-band DRC filter bank as specified in 6.4.11, only the “crossoverFreqIndex-syntax” is allowed. Note that if the “startSubBandIndex-syntax” is used, the frequency smoothing/fading with overlap weights according to Table 24 is not applied.
crossoverFreqIndex	See Table A.8.
startSubBandIndex	zero-based sub-band index for an available sub-band domain. The field is used to signal the start index for a specific DRC band.

With:

Table A.16 — Coding of metadata in drcCoefficientsBasic(), drcCoefficientsUniDrc(), and drcCoefficientsUniDrcV1()

Metadata field	Description
drcLocation	See Table 2.
drcFrameSizePresent	A value of 1 indicates that *bsDrcFrameSize* value is present.
bsDrcFrameSize	Encoding of DRC frame size according to Table A.17.
drcCharacteristicLeftPresent, drcCharacteristicRightPresent	Flag to indicate if a characteristic is defined (1) or not (0).
characteristicLeftCount, characteristicRightCount	Number of defined characteristics.
characteristicFormat	Format of characteristic: sigmoidal (0), segment-wise (1) representation.
bsGainLeft, bsGainRight	Encoded DRC gain according to Table AMD1.40 and Table AMD1.41.
bsIoRatioLeft, bsIoRatioRight	Encoded I/O ratio for sigmoidal characteristic according to Table AMD1.42.
bsExpLeft, bsExpRight	Encoded exponent for sigmoidal characteristic according to Table AMD1.43.
flipSignLeft, flipSignRight	Flag to indicate if the DRC characteristic is multiplied by -1 (1) or not (0).
bsCharNodeCount	Encoded node count for segment-wise representation according to Table AMD1.44.
bsNodeLevelDelta	Encoded differential DRC gain for this node according to Table AMD1.45.
bsNodeGain	Encoded DRC gain for this node according to Table AMD1.46.
shapeFiltersPresent	A value of 1 indicates that shape filter parameters are present.
shapeFilterCount	Number of shape filter parameter sets.
lfCutFilterPresent, lfBoostFilterPresent, hfCutFilterPresent, hfBoostFilterPresent,	A value of 1 indicates that the corresponding shape filter parameters are present.
lfCornerFreqIndex, hfCornerFreqIndex	Index of corner frequency parameter for low-frequency and high-frequency shape filters, respectively.
lfFilterStrengthIndex, hfFilterStrengthIndex,	Index of filter strength parameter for low-frequency and high-frequency shape filters, respectively.
gainSequenceCount	Number of gain sequences in the uniDrcGain() payload.
gainSetCount	Number of gain sequence sets. A set for multi-band DRC can include multiple gain sequences.
gainCodingProfile	See Table A.18.
gainInterpolationType	See Table A.19.
fullFrame	A value of 1 signals that the last node is always at the end of the frame. In low-delay mode, it shall be 1.
timeAlignment	A bit field that indicates whether the gain sample is aligned with the end (0) or the center (1) of the *deltaTmin* interval.
delayMode	A bit field that indicates whether the received gains are applied immediately or with a delay of one frame (see Table A.20).
timeDeltaMinPresent	A value of 1 indicates that a *bsTimeDeltaMin* value is present.
bsTimeDeltaMin	This field indicates the custom time resolution of a DRC sequence which overrides the default *deltaTmin* value. The values are encoded according to Table A.21.
bandCount	The number of DRC bands for this gain set.

Table A.16 *(continued)*

Metadata field	Description
drcBandType	A bit field, which signals if the "crossoverFreqIndex-syntax" (1) or the "startSubBandIndex-syntax" (0) should be used. If multi-band DRC gains are applied in the time-domain by using the multi-band DRC filter bank like specified in 6.4.11, only the "crossoverFreqIndex-syntax" is allowed. Note that if the "startSubBandIndex-syntax" is used, the frequency smoothing/ fading with overlap weights according to Table 24 is not applied.
indexPresent	A value of 1 indicates that a *bsIndex* value is present.
bsIndex	Encoded gain sequence index that is copied into *gainSequenceIndex* which refers to the gain sequences in uniDrcGain() numbered in the order of appearance.
drcCharacteristicPresent	Flag indicating whether a source characteristic for the gain sequence is present (1) or not (0).
drcCharacteristicFormatIsCICP	Flag indicating whether the index of the characteristic is based on ISO/ IEC 23008-1, (1) or not (0).
drcCharacteristic	A value of 0 means that the DRC characteristic is undefined for a DRC sequence. Values > 11 are reserved. See ISO/IEC 23001-8 (CICP).
drcCharacteristicLeftIndex, drcCharacteristicRightIndex	Index of source DRC characteristic of the gain sequence referring to *characteristicLeftIndex* and *characteristicRightIndex* in drcCoefficientsUniDrcV1().
crossoverFreqIndex	See Table A.8.
startSubBandIndex	Zero-based sub-band index for an available sub-band domain. The field is used to signal the start index for a specific DRC band.

Page 68, A.6.8

Replace Table A.24:

Table A.24 — Coding of metadata in drcInstructionsBasic() and drcInstructionsUniDrc()

Metadata field	Description
bsSequenceIndex	A unique number which specifies which DRC sequence should be assigned to which channel/object of the configuration specified in downmixId. The reserved value 0x00 indicates that the assigned channel will not be processed by the DRC tool. All other values indicate the assigned DRC sequence index plus one. Due to the reserved value, the number of DRC gain sequences in one location cannot be more than 63. Note that *bsSequenceIndex* is effectively a 1-based index. The sequence indices (*sequenceIndex*) used for processing are 0-based.
additionalDownmixIdCount	Number of additional downmixIds connected with one DRC set. Note that only one DRC channel group is allowed if an additional downmixId refers to a different target channel layout than the first downmixId.
additionalDownmixId	Additional downmixId connected with a drcSetId. This field may be used to declare additional target layouts for which one single DRC set provides suitable gains, e.g. for clipping prevention of multiple target layouts with the same DRC set. Note that only one DRC channel group is allowed if an additional downmixId refers to a different target channel layout than the first downmixId.

Table A.24 *(continued)*

Metadata field	Description
bsRepeatSequenceCount	A field which declares that the current sequence index should be repeated for multiple channels. The assignment for-loop continues at position "i = i + repeatSequenceCount". bsRepeatSequenceCount is encoded according to Table A.26. If a sequence index should be repeated more than 32 times, bsSequenceIndex has to be signalled again.
bsRepeatParametersCount	A field similar to bsRepeatSequenceCount to indicate if the sequence index and the duckingScaling factor should be repeated for multiple channels. bsRepeatParametersCount is encoded according to Table A.26.
noIndependentUse	A flag which signals that the DRC set can only be used in combination with another DRC set as indicated by the *dependsOnDrcSet* field.
bsDrcSetTargetLoudness-ValueUpper	A field which contains the upper limit of the target loudness of a DRC set. The default value is 0 dB. The values are encoded according to Table A.31.
bsDrcSetTargetLoudness-ValueLower	A field which contains the lower limit of the target loudness of a DRC set. The default value is -63 dB. The values are encoded according to Table A.31.

With:

Table A.24 — Coding of metadata in drcInstructionsBasic(),drcInstructionsUniDrc(), and drcInstructionsUniDrcV1()

Metadata field	Description
drcSetId	Index to identify this DRC set. The value of 0 and 0×3F is reserved.
drcSetComplexityLevel	Value that indicates the computational complexity level of the associated DRC set per audio channel. The complexity level is used to determine if a decoder is capable of applying the DRC set (see 6.9).
drcLocation	Location of DRC gain sequence (see Table 2).
downmixIdPresent	A flag that has a value of 1 when *downmixID* is present.
downmixId	Index to identify a downmix that is associated with this DRC set.
additionalDownmixIdPresent	A flag that has a value of 1 if additional *downmixIds* are present.
additionalDownmixIdCount	Number of additional *downmixIds* connected with one DRC set. Note that only one DRC channel group is allowed if an additional downmixId is present and the DRC is applied to a downmix.
additionalDownmixId	Additional downmixId connected with a drcSetId. This field may be used to declare additional target layouts for which one single DRC set provides suitable gains, e.g. for clipping prevention of multiple target layouts with the same DRC set. Note that only one DRC channel group is allowed if an additional downmixId is present and the DRC is applied to a downmix.
drcApplyToDownmix	A flag that has a value of 1 to indicate that the DRC set is applied to the downmix. Otherwise it is applied to the base layout. Since this flag is only present in drcIntructionsUniDrcV1() payloads, for the other payloads the DRC is applied to the downmix if a *downmixID* is present that is different from 0×0.
drcSetEffect	Declares the effect of the DRC according to Table A.32.
limiterPeakTargetPresent	A flag that has a value of 1 if a *bsLimiterPeakTarget* value is present.
bsLimiterPeakTarget	Encoded limiter peak target value according to Table A.27.
drcSetTargetLoudnessPresent	A flag that has a value of 1 if a *bsDrcSetTargetLoudnessValueUpper* value is present.
bsDrcSetTargetLoudness-ValueUpper	A field which contains the upper limit of the target loudness of a DRC set. The default value is 0 dB. The values are encoded according to Table A.31.
bsDrcSetTargetLoudnessValue-LowerPresent	A flag that has a value of 1 if a *bsDrcSetTargetLoudnessValueLower* value is present.

Table A.24 *(continued)*

Metadata field	Description
bsDrcSetTargetLoudnessValue-Lower	A field which contains the lower limit of the target loudness of a DRC set. The default value is -63 dB. The values are encoded according to Table A.31.
dependsOnDrcSetPresent	A flag that has a value of 1 if a *dependsOnDrcSet* value is present.
dependsOnDrcSet	Indicates the drcSetId of the DRC set this DRC depends on. Same encoding as drcSetId.
noIndependentUse	A flag which signals that the DRC set can only be used in combination with another DRC set as indicated by the *dependsOnDrcSet* field.
requiresEq	A flag which signals that the DRC set can only be used in combination with an EQ set, i.e. if *requiresEq*==1 it is not permitted to use this DRC set without an associated EQ set, and a DRC tool that does not support EQ shall ignore this DRC set. For combined DRC sets with a primary and dependent set, the *requiresEq* field of the dependent set is ignored.
duckingScalingPresent	A flag that has a value of 1 if a *bsDuckingScaling* value is present.
bsDuckingScaling	Encoded scaling value for ducking.
repeatParameters	A flag that has a value of 1 if a *bsRepeatParametersCount* value is present.
bsRepeatParametersCount	Encoded count value indicating how many times the same *bsGainSetIndex* shall be repeatedly used for subsequent channels (see Table A.26).
bsGainSetIndex	A unique number which specifies which DRC gain set should be assigned to which channel/object of the configuration specified in downmixId. The reserved value 0×00 indicates that the assigned channel will not be processed by the DRC tool. All other values indicate the assigned DRC gain set index according to gainSetIndex. Due to the reserved value, the number of DRC gain sets in one location cannot be more than 63.
repeatGainSetIndex	Flag to indicate whether the gain set index is repeated (1) or not (0).
bsRepeatGainSetIndexCount	A field which declares that the current gain set index should be repeated for multiple channels. The assignment for-loop continues at position "i = i + *repeatGainSetCount*". *bsRepeatGainSetIndexCount* is encoded according to Table A.26. If a sequence index should be repeated more than 32 times, *bsGainSetIndex* has to be signaled again.
targetCharacteristicLeftPresent, targetCharacteristicRightPresent	Flag to indicate whether a target DRC characteristic is defined (1) or not (0).
targetCharacteristicLeftIndex, targetCharacteristicRightIndex	Index of left and right target DRC characteristic, respectively. The index refers to *characteristicLeftIndex* and *characteristicRightIndex*, in drcCoefficientsUniDrcV1(), respectively. The index value of 0 is reserved.
gainScalingPresent	A flag that has a value of 1 if a scaling values are present.
bsAttenuationScaling, bsAmplificationScaling	Encoded scaling values according to Table A.29.
gainOffsetPresent	A flag that has a value of 1 if an offset value is present.
bsGainOffset	Encoded offset value according to Table A.30.
shapeFilterPresent	A value of 1 indicates that a shape filter index is present.
shapeFilterIndex	1-based index that refers to a specific shape filter parameter set in the drcCoefficientsUniDrcV1() payload as indicated by the *shapeFIlterIndex* value in Table AMD1.24.

Page 65, A.6.8

Replace Table A.25:

Table A.25 — Coding of bsSequenceIndex

Encoding	Size	Mnemonic	sequenceIndex	Range
μ	6 bits	uimsbf	$\mu - 1$ if $\mu > 0$, else undefined	0...62

With:

Table A.25 — Coding of bsGainSetIndex

Encoding	Size	Mnemonic	gainSetIndex	Range
μ	6 bits	uimsbf	$\mu - 1$ if $\mu > 0$, else undefined	0...62

Page 68, A.6.8

Replace Table A.26:

Table A.26 — Coding of bsRepeatParametersCount and bsRepeatSequenceCount field

Encoding	Size	Mnemonic	repeatParametersCount, repeatSequenceCount	Range
μ	5 bits	uimsbf	$N_R = \mu + 1$	1...32

With:

Table A.26 — Coding of bsRepeatParametersCount and bsRepeatGainSetIndexCount field

Encoding	Size	Mnemonic	repeatParametersCount, repeatGainSetIndexCount	Range
μ	5 bits	uimsbf	$N_R = \mu + 1$	1...32

Page 67, A.6.7

Add new tables after Table A.23 and adjust subsequent table numbering:

Table AMD1.40 — Coding of bsGainRight

Encoding	Size	Mnemonic	gainDbRight [dB]	Range
μ	6 bits	uimsbf	$-\mu$	-63...0 dB, 1 dB step size

Table AMD1.41 — Coding of bsGainLeft

Encoding	Size	Mnemonic	gainDbLeft [dB]	Range
μ	6 bits	uimsbf	μ	0...63 dB, 1 dB step size

Table AMD1.42 — Coding of bsIoRatioLeft and bsIoRatioRight

Encoding	Size	Mnemonic	ioRatioLeft, ioRatio-Right	Range
μ	4 bits	uimsbf	$0.05 + 0.15\mu$	0.05...2.3, step size of 0.15

Table AMD1.43 — Coding of bsExpLeft and bsExpRight

Encoding	Size	Mnemonic	expLeft, expRight	Range
μ	4 bits	uimsbf	$1 + 2\mu$; $\mu < 15$ ∞ ; else	1...29, step size of 2 and ∞

Table AMD1.44 — Coding of bsCharNodeCount

Encoding	Size	Mnemonic	characteristicNodeCount	Range
μ	2 bits	uimsbf	$\mu + 1$	1...4

Table AMD1.45 — Coding of bsNodeLevelDelta

Encoding	Size	Mnemonic	nodeLevelDelta [dB]	Range
μ	5 bits	uimsbf	$\mu + 1$	1...32 dB, 1 dB step size

Table AMD1.46 — Coding of bsNodeGain

Encoding	Size	Mnemonic	nodeGain [dB]	Range
μ	8 bits	uimsbf	$0.5\mu - 64$	-64...63.5 dB, 0.5 dB step size

Table AMD1.47 — Coding of coefficient boundary $y_{1,bound}$ for LF shaping filters depending on lfCornerFreqIndex and lfFilterStrengthIndex in drcCoefficientsUniDrcV1 ()

lfCornerFreqIndex	lfFilterStrengthIndex			
	0	1	2	3 (reserved)
0	-0.994	-0.996	-1.0	*reserved*
1	-0.99	-0.995	-0.999	*reserved*
2	-0.98	-0.989	-0.996	*reserved*
3	-0.97	-0.983	-0.994	*reserved*
Remaining indexes reserved	*reserved*	*reserved*	*reserved*	*reserved*

Table AMD1.48 — Coding of coefficient boundary $y_{1,bound}$ for HF shaping filters depending on hfCornerFreqIndex and hfFilterStrengthIndex in drcCoefficientsUniDrcV1 ()

hfCornerFreqIndex	hfFilterStrengthIndex			
	0	1	2	3 (reserved)
0	0.15	0.75	1.05	*reserved*
1	0.43	0.87	1.07	*reserved*
2	0.6	0.92	1.07	*reserved*
3	0.8	1.0	1.06	*reserved*
4	0.9	1.04	1.073	*reserved*
Remaining indexes reserved	*reserved*	*reserved*	*reserved*	*reserved*

Table AMD1.49 — Coding of coefficient gain offset g_{offset} for LF shaping filters depending on lfCornerFreqIndex and lfFilterStrengthIndex in drcCoefficientsUniDrcV1 ()

lfCornerFreqIndex	lfFilterStrengthIndex			
	0	1	2	3 (reserved)
0	3.0	2.0	1.2	*reserved*
1	3.0	2.0	1.5	*reserved*
2	3.0	2.0	2.0	*reserved*
3	3.0	2.0	2.0	*reserved*
Remaining indexes reserved	*reserved*	*reserved*	*reserved*	*reserved*

Table AMD1.50 — Coding of coefficient gain offset g_{offset} for HF shaping filters depending on hfCornerFreqIndex and hfFilterStrengthIndex in drcCoefficientsUniDrcV1 ()

hfCornerFreqIndex	hfFilterStrengthIndex			
	0	1	2	3 (reserved)
0	4.5	6.0	3.5	*reserved*
1	3.7	4.0	2.7	*reserved*
2	3.0	3.5	2.0	*reserved*
3	2.0	2.5	1.5	*reserved*
4	1.5	2.0	1.31	*reserved*
Remaining indexes reserved	*reserved*	*reserved*	*reserved*	*reserved*

Table AMD1.51 — Coding of Radius *r* for LF shaping filters depending on lfCornerFreqIndex in drcCoefficientsUniDrcV1 ()

lfCornerFreqIndex	*r*
0	0.988
1	0.98
2	0.96
3	0.94
Remaining indexes reserved	*reserved*

Table AMD1.52 — Coding of normalized cutoff frequency $f_{c,norm}$ and radius *r* for HF shaping filters depending on hfCornerFreqIndex in drcCoefficientsUniDrcV1 ()

hfCornerFreqIndex	$f_{c,norm}$	*r*
0	0.15	0.45
1	0.20	0.4
2	0.25	0.35
3	0.35	0.3
4	0.45	0.3
Remaining indexes reserved	*reserved*	*reserved*

Page 69, A.6.8

Replace Table A.31:

Table A.31 — Coding of bsDrcSetTargetLoudnessValueUpper/-Lower field

Encoding	Size	Mnemonic	drcSetTargetLoudnessValue in [dBFS]	Range
μ	6 bits	uimsbf	drcSetTargetLoudnessValue = μ − 63	-63 ... 0 dBFS, 1 dB step size.

With:

Table A.31 — Coding of bsDrcSetTargetLoudnessValueUpper/-Lower field

Encoding	Size	Mnemonic	drcSetTargetLoudnessValue in [LKFS]	Range
μ	6 bits	uimsbf	drcSetTargetLoudnessValue = μ − 63	-63 ... 0 LKFS, 1 dB step size.

Page 71, A.6.9

Replace Table A.35:

Table A.35 — Coding of methodValue field

Encoding	Format	methodDefinition	methodValue	Approx. range
μ	8 uimsbf	1, ..., 5	L = -57.75 + $\mu 2^{-2}$	-57.75 ... 6, 0.25 step size
See Table A.39	8 uimsbf	6	See Table A.39	0 ... 121
μ	5 uimsbf	7	L = 80 + μ	80 ... 111 dB
μ	2 uimsbf	8	0×0: ”not indicated” 0×1: ”large room, X curve monitor”[1] 0×2: ”small room, flat monitor”[1] 0×3: “reserved”	n/a
μ	8 uimsbf	9	L = -116 + $\mu 2^{-1}$	-116 ... 11.5 LKFS 0.5 step size

With:

Table A.35 — Coding of methodValue field

Encoding	Format	methodDefinition	methodValue	Approx. range
μ	8 uimsbf	0, ..., 5	$L = -57.75 + \mu 2^{-2}$	-57.75 ... 6, 0.25 step size
See Table A.39	8 uimsbf	6	See Table A.39	0 ... 121
μ	5 uimsbf	7	$L = 80 + \mu$	80 ... 111 dB
μ	2 uimsbf	8	0×0: "not indicated" 0×1: "large room, X curve monitor"[1] 0×2: "small room, flat monitor"[1] 0×3: "reserved"	n/a
μ	8 uimsbf	9	$L = -116 + \mu 2^{-1}$	-116 ... 11.5 LKFS, 0.5 step size

Page 71, A.6.9

Replace Table A.37 with:

Table A.37 — Coding of measurementSystem field in loudnessInfo()

Value	Meaning
0	Unknown/other
1	EBU R-128[8]
2	ITU-R BS.1770-4[4]
3	ITU-R BS.1770-4 with pre-processing. The pre-processor is a 4th order Linkwitz-Riley filter with a cutoff frequency of 500 Hz.
4	User
5	Expert panel
6	ITU-R BS.1771-1[5]
7 *(reserved, not permitted)*	Reserved Measurement System A (RMS_A)
8 *(reserved, not permitted)*	Reserved Measurement System B (RMS_B)
9 *(reserved, not permitted)*	Reserved Measurement System C (RMS_C)
10 *(reserved, not permitted)*	Reserved Measurement System D (RMS_D)
11 *(reserved, not permitted)*	Reserved Measurement System E (RMS_E)
Remaining values are reserved.	

Page 73, A.6.9

Add new subclauses after A.6.9:

A.6.10 Coded metadata in drcCoefficientsParametricDrc()

Table AMD1.53 — Coding of metadata in drcCoefficientsParametricDrc()

Metadata field	Description
drcLocation	See 6.1.2.3. If a drcCoefficientsUniDrc() block for the same *drcLocation* is present, the first *gainSetIndex* referring to drcCoefficientsParametricDrc() starts with an offset of the *gainSetCount* defined in drcCoefficientsUniDrc(). Note that the same holds for a drcCoefficientsUniDrcV1() block if present.
parametricDrcFrameSizeFormat	A field that signals whether parametricDrcFrameSize is coded by *bsParametricDrcFrameSize* (0) or by *bsDrcFrameSize* (1).
bsParametricDrcFrameSize	A field that signals the processing frame size of parametric DRCs. The values are encoded according to Table AMD1.54.
bsDrcFrameSize	A field that signals the processing frame size of parametric DRCs. The values are encoded according to Table A.17.
bsParametricDrcDelayMax	A field that specifies a delay value greater than or equal to the maximum delay of any applicable parametric DRC set combination in the stream. The values are encoded according to Table AMD1.55.
resetParametricDrc	The default value is 0. A value of 1 indicates that the parametric DRC shall be reset (e.g. for streaming scenarios). See also 6.6.
parametricDrcGainSetCount	Number of parametric DRC gain sets.
parametricDrcId	A unique 0-based identifier that refers to a specific parametricDrcInstructions() block.
sideChainConfigType	A field that defines the side-chain configuration of a parametric DRC gain set. The values are encoded according to Table AMD1.56. For a value of 1, the channel weights can be customized based on an applicable *downmixId*. For all other values, the side-chain input signal is defined by the *drcChannelGroup* the DRC gains set is assigned to within a drcInstructionsUniDrc() or drcInstructionsUniDrcV1() block. If a value is not applicable for a parametric DRC instance, sideChainConfigType shall be set to 0, which is the default. Note that for drcInstructionsUniDrcV1(), the applicable *downmixId* is 0 (baseLayout) for *drcApplyToDownmix*=0.
downmixId	See Table A.13.
levelEstimChannelWeightFormat	A field that signals if a channel map (0) or a channel weight map (1) is present for the definition of the side-chain signal of a parametric DRC instance.
addChannel	A value of 1 indicates that the current channel shall be added to the side-chain signal of a parametric DRC instance.
bsChannelWeight	A field that defines a weighting factor for the current channel for addition to the side-chain signal of a parametric DRC instance. The values are encoded according to Table AMD1.57.
bsDrcInputLoudness	A field that signals the input loudness for a parametric DRC gain set. The values are encoded according to Table AMD1.58. The input loudness is required for normalization of the DRC side-chain input level.

Table AMD1.54 — Coding of bsParametricDrcFrameSize field

Field	Encoding	Mnemonic	parametricDrcFrameSize in [samples]	Range
bsParametricDrcFrameSize	μ 4 bits	uimsbf	$M_{parametricDRC} = 2\mu$	1...2^{15} audio sample intervals, variable step size.

Table AMD1.55 — Coding of bsParametricDrcDelayMax field

Field	Encoding	Mnemonic	parametricDrcDelayMax in intervals of audio sample rate	Range
bsParametricDrcDelayMax	$\{\mu, v\}$ {5 bits, 3 bits}	{uimsbf, uimsbf}	$16\mu 2^{v}$	0...63488, variable step size.

Table AMD1.56 — Coding of sideChainConfigType field

Value	Meaning
0 (default)	Side-chain signal defined by all channels of the processed *drcChannelGroup* and *channelWeight*[] set to 0 dB.
1	Custom configuration of side-chain signal based on an applicable *downmixId*.
2	Side-chain signal defined by all channels of the processed *drcChannelGroup* and *channelWeight*[] set according to ITU-R BS.1770-4.
Remaining values are reserved.	

Table AMD1.57 — Coding of bsChannelWeight field

Field	Encoding	Mnemonic	μ	channelWeight	Range
bsChannelWeight	μ 4 bits	uimsbf	0	10.0 dB	-∞ ... 10.0 dB, variable step size.
			1	6.0 dB	
			2	4.5 dB	
			3	3.0 dB	
			4	1.5 dB	
			5	0 dB	
			6	-1.5 dB	
			7	-3.0 dB	
			8	-4.5 dB	
			9	-6.0 dB	
			10	-10.0 dB	
			11	-15.0 dB	
			12	-20.0 dB	
			13	-30.0 dB	
			14	-40.0 dB	
			15	-∞ dB	

Table AMD1.58 — Coding of bsDrcInputLoudness field

Field	Encoding	Mnemonic	drcInputLoudness in [dB]	Range
bsDrcInputLoudness	μ 8 bits	uimsbf	$L_{reference} = -57.75 + \mu 2^{-2}$	-57.75 ... 6 dB, 0.25 dB step size.

A.6.11 Coded metadata in parametricDrcInstructions()

Table AMD1.59 — Coding of metadata in parametricDrcInstructions()

Metadata field	Description
parametricDrcId	A unique 0-based identifier for each parametricDrcInstructions() block.
bsParametricDrcLookAhead	A field that signals the audio delay that should be applied for a parametric DRC instance. See Table AMD1.60. If not present, a default value is set based on *parametricDrcType*.
parametricDrcPresetIdPresent	A field that indicates whether the parametric DRC type is defined by *parametricDrcPresetId* (1) or not (0).
parametricDrcPresetId	A unique 0-based identifier that is mapped to a *parametricDrcType* and predefined settings according to Table AMD1.62. If an unsupported value is received, the corresponding parametric DRC instance shall be disabled.
parametricDrcType	A field that signals the parametric DRC type. See Table AMD1.61.

Table AMD1.60 — Coding of bsParametricDrcLookAhead field

Field	Encoding	Mnemonic	parametricDrcLookAhead in [ms]	Range
bsParametricDrcLookAhead	μ 7 bits	uimsbf	$T_{lookAhead} = \mu$	0...127 ms, 1 ms step size.

Table AMD1.61 — Coding of parametricDrcType

Symbol	Value of parametricDrcType	Purpose
PARAM_DRC_TYPE_FF	0×0	Parameters defined by parametricDrcTypeFeedForward() block (see 6.6.2.3 and 6.6.3.1).
PARAM_DRC_TYPE_LIM	0×1	Parameters defined by parametricDrcTypeLimiter() block (see 6.6.2.4 and 6.6.3.2).
(reserved)	*2-6*	For ISO use.
(reserved)	*7*	For use outside of ISO scope.

Table AMD1.62 — Mapping of parametricDrcPresetId to parametricDrcType

parametricDrcPresetId	parametricDrcType	Parameters	Description
0	PARAM_DRC_TYPE_FF	drcCharacteristic = 7	DRC curve definition according ISO/IEC 23001-8. DRC gain smoothing time constants according to Table AMD1.77.
1	PARAM_DRC_TYPE_FF	drcCharacteristic = 8	
2	PARAM_DRC_TYPE_FF	drcCharacteristic = 9	
3	PARAM_DRC_TYPE_FF	drcCharacteristic = 10	
4	PARAM_DRC_TYPE_FF	drcCharacteristic = 11	
5	PARAM_DRC_TYPE_LIM	—	Default parameters as specified in 6.6.3.2.
6-127	*(reserved)*	*(reserved)*	*(reserved)*

A.6.12 Coded metadata in parametricDrcTypeFeedForward()

Table AMD1.63 — Coding of metadata in parametricDrcTypeFeedForward()

Metadata field	Description
levelEstimKWeightingType	A field that indicates the state of the pre-filter and RLB filter for level estimation according to ITU-R BS.1770-4. See Table AMD1.64.
bsLevelEstimIntegrationTime	Level estimation integration time. If not present, the default value is *parametricDrcFrameSize*. See Table AMD1.65.
drcCurveDefinitionType	A field that indicates whether the DRC curve is defined by a *drcCharacteristic* index (0) or by a DRC curve parametrization.
drcCharacteristic	*drcCharacteristic* index and DRC curve definition according to ISO/IEC 23001-8. Following *drcCharacteristic* indices are currently supported for the usage with parametricDrcType == PARAM_DRC_TYPE_FF: {7,8,9,10,11}. If an unsupported value is received, the corresponding parametric DRC instance shall be disabled.
bsNodeCount	Number of curve nodes. See Table AMD1.66.
bsNodeLevelInitial	Input level of first node in dB. See Table AMD1.67.
bsNodeLevelDelta	Input level distance to next node in dB. See Table AMD1.68.
bsNodeGain	Node output gain in dB. See Table AMD1.69.
drcGainSmoothParametersPresent	A field that indicates whether custom smoothing parameters are provided (1) or whether smoothing parameters matching to a present *drcCharacteristic* index shall be used (0) (see Table AMD1.77). If *drcCharacteristic* is not present, the default smoothing parameters shall be used.
bsGainSmoothAttackTimeSlow	Slow attack time for DRC gain smoothing in ms. See Table AMD1.70.
bsGainSmoothReleaseTimeSlow	Slow release time for DRC gain smoothing in ms. See Table AMD1.71.
bsGainSmoothAttackTimeFast	Fast attack time for DRC gain smoothing in ms. See Table AMD1.72.
bsGainSmoothReleaseTimeFast	Fast release time for DRC gain smoothing in ms. See Table AMD1.73.
bsGainSmoothAttackThreshold	Fast attack threshold for DRC gain smoothing in dB. See Table AMD1.74.
bsGainSmoothReleaseThreshold	Fast release threshold for DRC gain smoothing in dB. See Table AMD1.75.
bsGainSmoothHoldOff	Hold counter for DRC gain smoothing. See Table AMD1.76.

Table AMD1.64 — Coding of levelEstimKWeightingType field

Value	Meaning
0	Pre-filter OFF, RLB filter OFF
1	Pre-filter OFF, RLB filter ON
2	Pre-filter ON, RLB filter ON
3	reserved

Table AMD1.65 — Coding of bsLevelEstimIntegrationTime field

Field	Encoding	Mnemonic	levelEstimIntegrationTime in [samples]	Range
bsLevelEstimIntegrationTime	μ 6 bits	uimsbf	$T_{integration} = (\mu + 1)M_{parametricDRC}$	1 ... 64 times $M_{parametricDRC}$ audio sample intervals.

Table AMD1.66 — Coding of bsNodeCount field

Field	Encoding	Mnemonic	nodeCount	Range
bsNodeCount	μ 3 bits	uimsbf	$N_{nodes} = \mu + 2$	2 ... 9.

Table AMD1.67 — Coding of bsNodeLevelInitial field

Field	Encoding	Mnemonic	nodeLevelInitial in [dB]	Range
bsNodeLevelInitial	μ 6 bits	uimsbf	$L_{nodes,initial} = -11 - \mu$	-74...-11 dB, 1 dB step size.

Table AMD1.68 — Coding of bsNodeLevelDelta field

Field	Encoding	Mnemonic	nodeLevelDelta in [dB]	Range
bsNodeLevelDelta	μ 5 bits	uimsbf	$\Delta L_{node} = 1 + \mu$	1...32 dB, 1 dB step size.

Table AMD1.69 — Coding of bsNodeGain field

Field	Encoding	Mnemonic	nodeGain in [dB]	Range
bsNodeGain	μ 6 bits	uimsbf	$G_{node} = \mu - 39$	-39...+24 dB, 1 dB step size.

Table AMD1.70 — Coding of bsGainSmoothAttackTimeSlow field

Field	Encoding	Mnemonic	gainSmoothAttackTimeSlow in [ms]	Range
bsGainSmoothAttackTimeSlow	μ 8 bits	uimsbf	$\tau_{attack,slow} = \mu 5$	0...1275 ms, 5 ms step size.

Table AMD1.71 — Coding of bsGainSmoothReleaseTimeSlow field

Field	Encoding	Mnemonic	gainSmoothReleaseTimeSlow in [ms]	Range
bsGainSmoothReleaseTimeSlow	μ 8 bits	uimsbf	$\tau_{release,slow} = \mu 40$	0...10200 ms, 40 ms step size.

Table AMD1.72 — Coding of bsGainSmoothAttackTimeFast field

Field	Encoding	Mnemonic	gainSmoothAttackTimeFast in [ms]	Range
bsGainSmoothAttackTimeFast	μ 8 bits	uimsbf	$\tau_{attack,fast} = \mu 5$	0...1275 ms, 5 ms step size.

Table AMD1.73 — Coding of bsGainSmoothReleaseTimeFast field

Field	Encoding	Mnemonic	gainSmoothReleaseTimeFast in [ms]	Range
bsGainSmoothReleaseTimeFast	μ 8 bits	uimsbf	$\tau_{release,fast} = \mu 20$	0...5100 ms, 20 ms step size.

Table AMD1.74 — Coding of bsGainSmoothAttackThreshold field

Field	Encoding	Mnemonic	gainSmoothAttackThreshold in [dB]	Range
bsGainSmoothAttackThreshold	μ 5 bits	uimsbf	$Thr_{attack,fast} = \begin{cases} \mu; \text{ if } \mu \neq 31 \\ \infty; \text{ else} \end{cases}$	0...30 dB and ∞, 1 dB step size.

Table AMD1.75 — Coding of bsGainSmoothReleaseThreshold field

Field	Encoding	Mnemonic	gainSmoothReleaseThreshold in [dB]	Range
bsGainSmoothReleaseThreshold	μ 5 bits	uimsbf	$Thr_{release,fast} = \begin{cases} \mu; \text{ if } \mu \neq 31 \\ \infty; \text{ else} \end{cases}$	0...30 dB and ∞, 1 dB step size.

Table AMD1.76 — Coding of bsGainSmoothHoldOff field

Field	Encoding	Mnemonic	gainSmoothHoldOff	Range
bsGainSmoothHoldOff	μ 7 bits	uimsbf	$C_{holdOff} = \mu$	0...127.