
**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*





COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2017, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

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: $(x \bmod y) = x - y \cdot \text{floor}(x/y)$

`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 precede 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.

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 <i>noIndependentUse</i> 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 <i>drcInstructionsUniDrcV1()</i> payloads	If EQ is not supported, DRC sets with <i>requiresEq</i> =1 are discarded.
8	The range of the target loudness specified for a DRC set has to include the requested decoder target loudness.	If <i>drcSetTargetLoudnessPresent</i> ==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 occurrences 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).

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)

<pre> channelGroupForChannel[i] = k; k++; } } } } nDrcChannelGroups = k; </pre>

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 δT_{min} . 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 δT_{min} . 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 (drcCoefficientsUniDrcVlPresent == 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);
}

```

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.

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.

$$g_{\text{warped}} = \text{warpGain}(g_{\text{DRC}}, \text{mode})$$

$$y_1 = \begin{cases} x_1 & ; \quad g_{\text{warped}} \leq 0 \\ x_1 + (y_{1,\text{bound}} - x_1) \frac{g_{\text{warped}}}{g_{\text{warped,max}}} & ; \quad 0 < g_{\text{warped}} < g_{\text{warped,max}} \\ y_{1,\text{bound}} & ; \quad \text{else} \end{cases}$$

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

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

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:

$$\text{abs}(g_{\text{DRC}} - g_{\text{DRC,last}}) > 0.0001 g_{\text{DRC,last}}$$

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

6.4.11.3 Low-frequency shaping filters

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

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

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:

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

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

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

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

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

$$x_2 = r^2$$

$$y_2 = x_2$$

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 *bsParametricDrcDelayMax* 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.

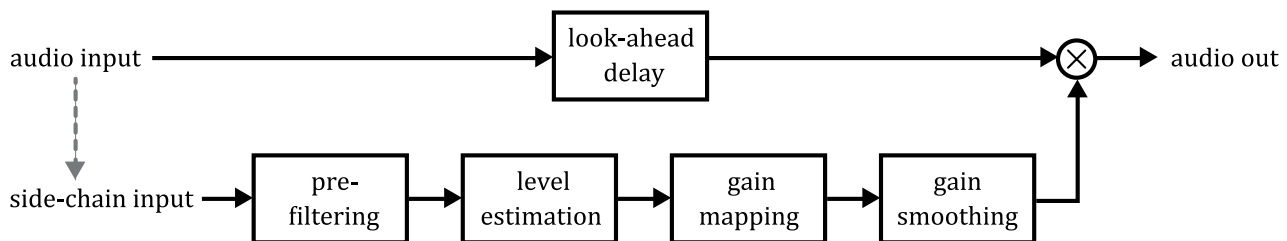


Figure AMD1.2 — Single-band feed-forward design of `parametricDrcType==0x0`

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:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}}$$

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(dsف=0; dsف<numDrcSubframes; dsف++)
{
    gainInDb = gainSmoothDb[dsف];
    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.

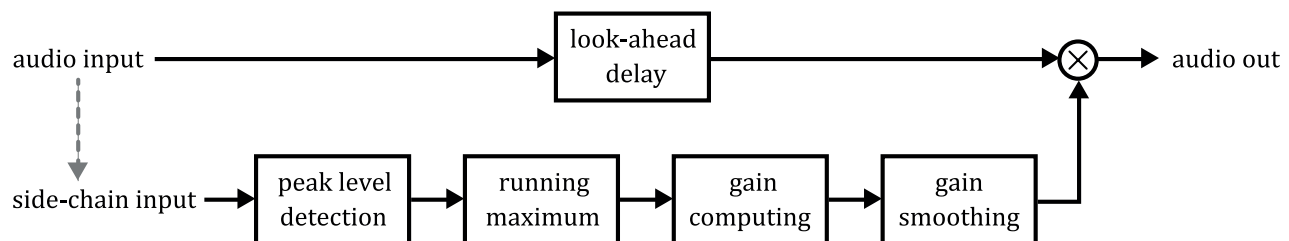


Figure AMD1.3 — Time domain peak limiter design of parametricDrcType==0x1

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.

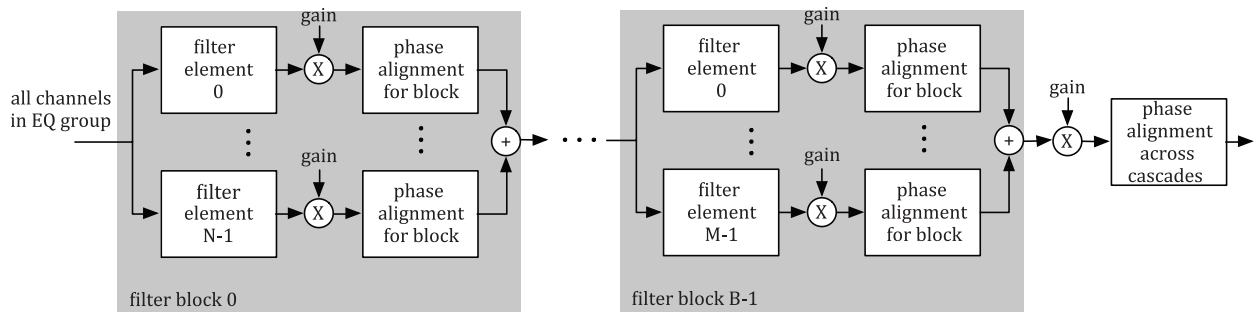


Figure AMD1.4 — Generic EQ filter cascade structure

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 or 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 auxiliary information of the payload describes the combined EQ including the dependent one. For the combined EQ, the auxiliary 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:

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

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 $|r| \neq 1$ 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.

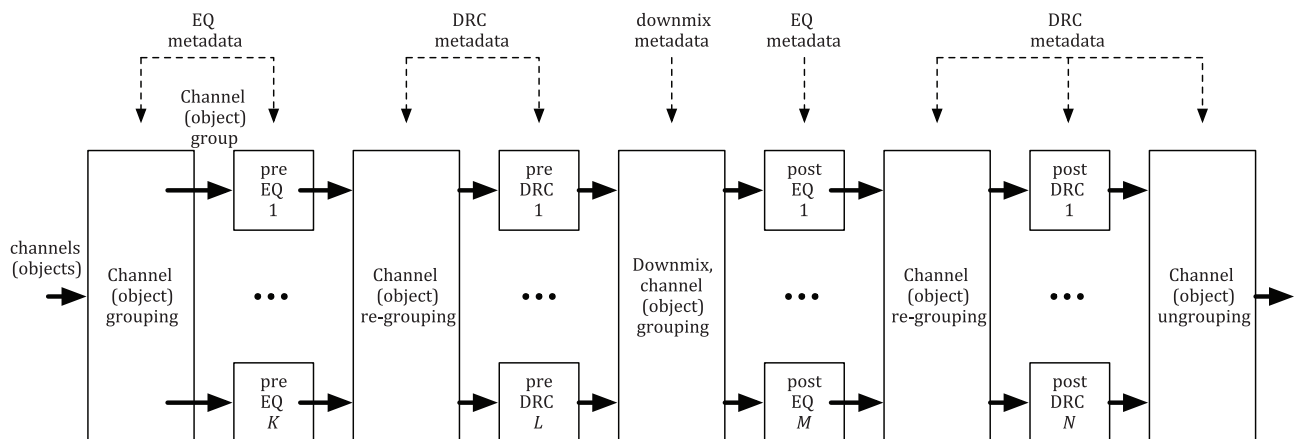


Figure AMD1.5 — Example overview of processing chain with EQ, DRC, and downmix

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:

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

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 <i>b</i> -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 $\omega = \omega_c(\text{bandIndex})$ in radians.