INTERNATIONAL STANDARD

ISO/IEC 14496-3

Third edition
2005-12-01

Information technology — Coding of audio-visual objects —

Part 3: **Audio**

Technologies de l'information — Codage des objets audiovisuels —

Partie 3: Codage audio

Reference number
ISO/IEC 14496-3:2005(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 14496-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This third edition cancels and replaces the second edition (ISO/IEC 14496-3:2001), which has been technically revised. It also incorporates the Amendments ISO/IEC 14496-3:2001/Amd.1:2003, ISO/IEC 14496-3:2001/Amd.2:2004, ISO/IEC 14496-3:2001/Amd.3:2005 and ISO/IEC 14496-3:2001/Amd.6:2005, and the Technical Corrigenda ISO/IEC 14496-3:2001/Cor.1:2002 and ISO/IEC 14496-3:2001/Cor.2:2004.

ISO/IEC 14496 consists of the following parts, under the general title *Information technology — Coding of audio-visual objects*:

— *Part 1: Systems*

— *Part 2: Visual*

— *Part 3: Audio*

— *Part 4: Conformance testing*

— *Part 5: Reference software*

— *Part 6: Delivery Multimedia Integration Framework (DMIF)*

— *Part 7: Optimized reference software for coding of audio-visual objects* [Technical Report]

— *Part 8: Carriage of ISO/IEC 14496 contents over IP networks*

— *Part 9: Reference hardware description* [Technical Report]

— *Part 10: Advanced Video Coding*

— *Part 11: Scene description and application engine*

— *Part 12: ISO base media file format*

— *Part 13: Intellectual Property Management and Protection (IPMP) extensions*

— *Part 14: MP4 file format*

— *Part 15: Advanced Video Coding (AVC) file format*

— *Part 16: Animation Framework eXtension (AFX)*

— *Part 17: Streaming text format*

— *Part 18: Font compression and streaming*

— *Part 19: Synthesized texture stream*

— *Part 20: Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF)*

0 Introduction

0.1 Overview

ISO/IEC 14496-3 (MPEG-4 Audio) is a new kind of audio standard that integrates many different types of audio coding: natural sound with synthetic sound, low bitrate delivery with high-quality delivery, speech with music, complex soundtracks with simple ones, and traditional content with interactive and virtual-reality content. By standardizing individually sophisticated coding tools as well as a novel, flexible framework for audio synchronization, mixing, and downloaded post-production, the developers of the MPEG-4 Audio standard have created new technology for a new, interactive world of digital audio.

MPEG-4, unlike previous audio standards created by ISO/IEC and other groups, does not target a single application such as real-time telephony or high-quality audio compression. Rather, MPEG-4 Audio is a standard that applies to *every* application requiring the use of advanced sound compression, synthesis, manipulation, or playback. The subparts that follow specify the state-of-the-art coding tools in several domains; however, MPEG-4 Audio is more than just the sum of its parts. As the tools described here are integrated with the rest of the MPEG-4 standard, exciting new possibilities for object-based audio coding, interactive presentation, dynamic soundtracks, and other sorts of new media, are enabled.

Since a single set of tools is used to cover the needs of a broad range of applications, *interoperability* is a natural feature of systems that depend on the MPEG-4 Audio standard. A system that uses a particular coder — for example a real-time voice communication system making use of the MPEG-4 speech coding toolset — can easily share data and development tools with other systems, even in different domains, that use the same tool — for example a voicemail indexing and retrieval system making use of MPEG-4 speech coding.

The remainder of this Introduction gives a more detailed overview of the capabilities and functioning of MPEG-4 Audio. First a discussion of concepts, that have changed since the MPEG-2 audio standards, is presented. Then the MPEG-4 Audio toolset is outlined.

0.2 Concepts of MPEG-4 Audio

As with previous MPEG standards, MPEG-4 does not standardize methods for encoding sound. Thus, content authors are left to their own decisions as to the best method of creating bitstream payloads. At the present time, methods to automatically convert natural sound into synthetic or multi-object descriptions are not mature; therefore, most immediate solutions will involve interactively-authoring the content stream in some way. This process is similar to current schemes for MIDI-based and multi-channel mixdown authoring of soundtracks.

Many concepts in MPEG-4 Audio are different from those in previous MPEG Audio standards. For the benefit of readers who are familiar with MPEG-1 and MPEG-2, we provide a brief overview here.

0.2.1 Audio storage and transport facilities

In all of the MPEG-4 tools for audio coding, the coding standard ends at the point of constructing access units that contain the compressed data. The MPEG-4 Systems (ISO/IEC 14496-1) specification describes how to convert these individually coded access units into elementary streams.

There is no standard transport mechanism of these elementary streams over a channel. This is because the broad range of applications that can make use of MPEG-4 technology have delivery requirements that are too wide to easily characterize with a single solution. Rather, what is standardized is an interface (the Delivery Multimedia Interface Format, or DMIF, specified in ISO/IEC 14496-6) that describes the capabilities of a transport layer and the communication between transport, multiplex, and demultiplex functions in encoders and decoders. The use of DMIF and the MPEG-4 Systems specification allows transmission functions that are much more sophisticated than are possible with previous MPEG standards.

However, LATM and LOAS were defined to provide a low overhead audio multiplex and transport mechanism for natural audio applications, which do not require sophisticated object-based coding or other functions provided by MPEG-4 Systems.

The following table gives an overview about the multiplex, storage and transmission formats currently available for MPEG-4 Audio within the MPEG-4 framework:

	Format	Functionality defined in MPEG-4:	Functionality originally defined in:	Description
Multiplex	M4Mux	ISO/IEC 14496-1 (normative)	-	MPEG-4 Multiplex scheme
	LATM	ISO/IEC 14496-3 (normative)	-	Low Overhead Audio Transport Multiplex
Storage	ADIF	ISO/IEC 14496-3 (informative)	ISO/IEC 13818-7 (normative)	Audio Data Interchange Format, (AAC only)
	MP4FF	ISO/IEC 14496-12 (normative)	-	MPEG-4 File Format
Transmission	ADTS	ISO/IEC 14496-3 (informative)	ISO/IEC 13818-7 (normative, exemplarily)	Audio Data Transport Stream, (AAC only)
	LOAS	ISO/IEC 14496-3 (normative, exemplarily)	-	Low Overhead Audio Stream, based on LATM, three versions are available: AudioSyncStream() EPAudioSyncStream() AudioPointerStream()

To allow for a user on the remote side of a channel to dynamically control a server streaming MPEG-4 content, MPEG-4 defines backchannel streams that can carry user interaction information.

0.2.2 MPEG-4 Audio supports low-bitrate coding

Previous MPEG Audio standards have focused primarily on transparent (undetectable) or nearly transparent coding of high-quality audio at whatever bitrate was required to provide it. MPEG-4 provides new and improved tools for this purpose, but also standardizes (and has tested) tools that can be used for transmitting audio at the low bitrates suitable for Internet, digital radio, or other bandwidth-limited delivery. The new tools specified in MPEG-4 are the state-of-the-art tools that support low-bitrate coding of speech and other audio.

0.2.3 MPEG-4 Audio is an object-based coding standard with multiple tools

Previous MPEG Audio standards provided a single toolset, with different configurations of that toolset specified for use in various applications. MPEG-4 provides several toolsets that have no particular relationship to each other, each with a different target function. The profiles of MPEG-4 Audio specify which of these tools are used together for various applications.

Further, in previous MPEG standards, a single (perhaps multi-channel or multi-language) piece of content was transmitted. In contrast, MPEG-4 supports a much more flexible concept of a *soundtrack*. Multiple tools may be used to transmit several *audio objects*, and when using multiple tools together an *audio composition* system is provided to create a single soundtrack from the several audio substreams. User interaction, terminal capability, and speaker configuration may be used when determining how to produce a single soundtrack from the component objects. This capability gives MPEG-4 significant advantages in quality and flexibility when compared to previous audio standards.

0.2.4 MPEG-4 Audio provides capabilities for synthetic sound

In natural sound coding, an existing sound is compressed by a server, transmitted and decompressed at the receiver. This type of coding is the subject of many existing standards for sound compression. In contrast, MPEG-4 standardizes a novel paradigm in which synthetic sound descriptions, including synthetic speech and synthetic

music, are transmitted and then *synthesized* into sound at the receiver. Such capabilities open up new areas of very-low-bitrate but still very-high-quality coding.

0.2.5 MPEG-4 Audio provides capabilities for error robustness

Improved error robustness capabilities for all coding tools are provided through the error-resilient bitstream payload syntax. This tool supports advanced channel coding techniques, which can be adapted to the special needs of given coding tools and a given communications channel. This error-resilient bitstream payload syntax is mandatory for all error resillient object types.

The error protection tool (EP tool) provides unequal error protection (UEP) for MPEG-4 Audio in conjunction with the error-resilient bitstream payload. UEP is an efficient method to improve the error robustness of source coding schemes. It is used by various speech and audio coding systems operating over error-prone channels such as mobile telephone networks or Digital Audio Broadcasting (DAB). The bits of the coded signal representation are first grouped into different classes according to their error sensitivity. Then error protection is individually applied to the different classes, giving better protection to more sensitive bits.

Improved error robustness for AAC is provided by a set of error resilience tools. These tools reduce the perceived degradation of the decoded audio signal that is caused by corrupted bits in the bitstream payload.

0.2.6 MPEG-4 Audio provides capabilities for scalability

Previous MPEG Audio standards provided a single bitrate, single bandwidth toolset, with different configurations of that toolset specified for use in various applications. MPEG-4 provides several bitrate and bandwidth options within a single stream, providing a scalability functionality that permits a given stream to scale to the requirement of different channels and applications or to be responsive to a given channel that has dynamic throughput characteristics. The tools specified in MPEG-4 are the state-of-the-art tools providing scalable compression of speech and audio signals.

0.3 The MPEG-4 Audio tool set

0.3.1 Speech coding tools

0.3.1.1 Overview

Speech coding tools are designed for the transmission and decoding of synthetic and natural speech.

Two types of speech coding tools are provided in MPEG-4. The *natural* speech tools allow the compression, transmission, and decoding of human speech, for use in telephony, personal communication, and surveillance applications. The *synthetic* speech tool provides an interface to text-to-speech synthesis systems; using synthetic speech provides very-low-bitrate operation and built-in connection with facial animation for use in low-bitrate video teleconferencing applications.

0.3.1.2 Natural speech coding

The MPEG-4 speech coding toolset covers the compression and decoding of natural speech sound at bitrates ranging between 2 and 24 kbit/s. When variable bitrate coding is allowed, coding at even less than 2 kbit/s, for example an average bitrate of 1.2 kbit/s, is also supported. Two basic speech coding techniques are used: One is a parametric speech coding algorithm, HVXC (Harmonic Vector eXcitation Coding), for very low bit rates; and the other is a CELP (Code Excited Linear Prediction) coding technique. The MPEG-4 speech coders target applications range from mobile and satellite communications, to Internet telephony, to packaged media and speech databases. It meets a wide range of requirements encompassing bitrate, functionality and sound quality.

MPEG-4 HVXC operates at fixed bitrates between 2.0 kbit/s and 4.0 kbit/s using a bitrate scalability technique. It also operates at lower bitrates, typically 1.2 - 1.7 kbit/s, using a variable bitrate technique. HVXC provides communications-quality to near-toll-quality speech in the 100 Hz – 3800 Hz band at 8 kHz sampling rate. HVXC also allows independent change of speed and pitch during decoding, which is a powerful functionality for fast access to speech databases. HVXC functionalities including 2.0 - 4.0 kbit/s fixed bitrate modes and a 2.0 kbit/s maximum variable bitrate mode.

Error Resilient (ER) HVXC extends operation of the variable bitrate mode to 4.0 kbit/s to allow higher quality variable rate coding. The ER HVXC therefore provides fixed bitrate modes of 2.0 - 4.0 kbit/s and a variable bitrate of either less than 2.0 kbit/s or less than 4.0 kbit/s, both in scalable and non-scalable modes. In the variable bitrate modes, non-speech parts are detected in unvoiced signals, and a smaller number of bits are used for these non-speech parts to reduce the average bitrate. ER HVXC provides communications-quality to near-toll-quality speech in the 100 Hz - 3800 Hz band at 8 kHz sampling rate. When the variable bitrate mode is allowed, operation at lower average bitrate is possible. Coded speech using variable bitrate mode at typical bitrates of 1.5 kbit/s average, and at typical bitrate of 3.0 kbit/s average has essentially the same quality as 2.0 kbit/s fixed rate and 4.0 kbit/s fixed rate respectively. The functionality of pitch and speed change during decoding is supported for all modes. ER HVXC has a bitstream payload syntax with the error sensitivity classes to be used with the EP-Tool, and some error concealment functionality is supported for use in error-prone channels such as mobile communication channels. The ER HVXC speech coder target applications range from mobile and satellite communications, to Internet telephony, to packaged media and speech databases.

MPEG-4 CELP is a well-known coding algorithm with new functionality. Conventional CELP coders offer compression at a single bit rate and are optimized for specific applications. Compression is one of the functionalities provided by MPEG-4 CELP, but MPEG-4 also enables the use of one basic coder in multiple applications. It provides scalability in bitrate and bandwidth, as well as the ability to generate bitstream payloads at arbitrary bitrates. The MPEG-4 CELP coder supports two sampling rates, namely, 8 kHz and 16 kHz. The associated bandwidths are 100 Hz – 3800 Hz for 8 kHz sampling and 50 Hz – 7000 Hz for 16 kHz sampling. The silence compression tool comprises a voice activity detector (VAD), a discontinuous transmission (DTX) unit and a comfort noise generator (CNG) module. The tool encodes/decodes the input signal at a lower bitrate during the non-active-voice (silent) frames. During the active-voice (speech) frames, MPEG-4 CELP encoding and decoding are used.

The silence compression tool reduces the average bitrate thanks to compression at a lower-bitrate for silence. In the encoder, a voice activity detector is used to distinguish between regions with normal speech activity and those with silence or background noise. During normal speech activity, the CELP coding is used. Otherwise a silence insertion descriptor (SID) is transmitted at a lower bitrate. This SID enables a comfort noise generator (CNG) in the decoder. The amplitude and the spectral shape of this comfort noise are specified by energy and LPC parameters in methods similar to those used in a normal CELP frame. These parameters are optionally re-transmitted in the SID and thus can be updated as required.

MPEG has conducted extensive verification testing in realistic listening conditions in order to prove the efficacy of the speech coding toolset.

0.3.1.3 Text-to-speech interface

Text-to-speech (TTS) capability is becoming a rather common media type and plays an important role in various multi-media application areas. For instance, by using TTS functionality, multimedia content with narration can be easily created without recording natural speech. Before MPEG-4, however, there was no way for a multimedia content provider to easily give instructions to an unknown TTS system. With **MPEG-4 TTS Interface**, a single common interface for TTS systems is standardized. This interface allows speech information to be transmitted in the international phonetic alphabet (IPA), or in a textual (written) form of any language.

The **MPEG-4 Hybrid/Multi-Level Scalable TTS Interface** is a superset of the conventional TTS framework. This extended TTS Interface can utilize prosodic information taken from natural speech in addition to input text and can thus generate much higher-quality synthetic speech. The interface and its bitstream payload format is scalable in terms of this added information; for example, if some parameters of prosodic information are not available, a decoder can generate the missing parameters by rule. Normative algorithms for speech synthesis and text-to-phoneme translation are not specified in MPEG-4, but to meet the goal that underlies the MPEG-4 TTS Interface, a decoder should fully utilize all the provided information according to the user's requirements level.

As well as an interface to text-to-speech synthesis systems, MPEG-4 specifies a joint coding method for phonemic information and facial animation (FA) parameters and other animation parameters (AP). Using this technique, a single bitstream payload may be used to control both the text-to-speech interface and the facial animation visual object decoder (see ISO/IEC 14496-2, Annex C). The functionality of this extended TTS thus ranges from conventional TTS to natural speech coding and its application areas, from simple TTS to audio presentation with TTS and motion picture dubbing with TTS.

0.3.2 Audio coding tools

0.3.2.1 Overview

Audio coding tools are designed for the transmission and decoding of recorded music and other audio soundtracks.

0.3.2.2 General audio coding tools

MPEG-4 standardizes the coding of natural audio at bitrates ranging from 6 kbit/s up to several hundred kbit/s per audio channel for mono, two-channel-, and multi-channel-stereo signals. General high-quality compression is provided by incorporating the MPEG-2 AAC standard (ISO/IEC 13818-7), with certain improvements, as MPEG-4 AAC. At 64 kbit/s/channel and higher ranges, this coder has been found in verification testing under rigorous conditions to meet the criterion of "indistinguishable quality" as defined by the European Broadcasting Union.

General audio (GA) coding tools comprise the AAC tool set expanded by alternative quantization and coding schemes (Twin-VQ and BSAC). The general audio coder uses a perceptual filterbank, a sophisticated masking model, noise-shaping techniques, channel coupling, and noiseless coding and bit-allocation to provide the maximum compression within the constraints of providing the highest possible quality. Psychoacoustic coding standards developed by MPEG have represented the state-of-the-art in this technology since MPEG-1 Audio; MPEG-4 General Audio coding continues this tradition.

For bitrates ranging from 6 kbit/s to 64 kbit/s per channel, the MPEG-4 standard provides extensions to the GA coding tools, that allow the content author to achieve the highest quality coding at the desired bitrate. Furthermore, various bit rate scalability options are available within the GA coder. The low-bitrate techniques and scalability modes provided within this tool set have also been verified in formal tests by MPEG.

The **MPEG-4 low delay** coding functionality provides the ability to extend the usage of generic low bitrate audio coding to applications requiring a very low delay in the encoding / decoding chain (e.g. full-duplex real-time communications). In contrast to traditional low delay coders based on speech coding technology, the concept of this low delay coder is based on general perceptual audio coding and is thus suitable for a wide range of audio signals. Specifically, it is derived from the proven architecture of MPEG-2/4 Advanced Audio Coding (AAC) and all capabilities for coding of 2 (stereo) or more sound channels (multi-channel) are available within the low delay coder. It operates at up to 48 kHz sampling rate and uses a frame length of 512 or 480 samples, compared to the 1024 or 960 samples used in standard MPEG-2/4 AAC to enable coding of general audio signals with an algorithmic delay not exceeding 20 ms. Also the size of the window used in the analysis and synthesis filterbank is reduced by a factor of 2. No block switching is used to avoid the "look-ahead'" delay due to the block switching decision. To reduce pre-echo artefacts in the case of transient signals, window shape switching is provided instead. For non-transient portions of the signal a sine window is used, while a so-called low overlap window is used for transient portions. Use of the bit reservoir is minimized in the encoder in order to reach the desired target delay. As one extreme case, no bit reservoir is used at all.

The **MPEG-4 BSAC** is used in combination with the AAC coding tools and replaces the noiseless coding of the quantized spectral data and the scalefactors. The MPEG-4 BSAC provides fine grain scalability in steps of 1 kbit/s per audio channel, i.e. 2 kbit/s steps for a stereo signal. One base layer stream and many small enhancement layer streams are used. To obtain fine step scalability, a bit-slicing scheme is applied to the quantized spectral data. First the quantized spectral values are grouped into frequency bands. Each of these groups contains the quantized spectral values in their binary representation. Then the bits of a group are processed in slices according to their significance. Thus all most significant bits (MSB) of the quantized values in a group are processed first. These bit-slices are then encoded using an arithmetic coding scheme to obtain entropy coding with minimal redundancy. In order to implement fine grain scalability efficiently using MPEG-4 Systems tools, the fine grain audio data can be grouped into large-step layers and these large-step layers can be further grouped by concatenating large-step layers from several sub-frames. Furthermore, the configuration of the payload transmitted over an Elementary Stream (ES) can be changed dynamically (by means of the MPEG-4 backchannel capability) depending on the environment, such as network traffic or user interaction. This means that BSAC can allow for real-time adjustments to the quality of service. In addition to fine grain scalablity, it can improve the quality of an audio signal that is decoded from a stream transmitted over an error-prone channel, such as a mobile communication networks or Digital Audio Broadcasting (DAB) channel.

MPEG-4 SBR (Spectral Band Replication) is a bandwidth extension tool used in combination with the AAC general audio codec. When integrated into the MPEG AAC codec, a significant improvement of the performance is available, which can be used to lower the bitrate or improve the audio quality. This is achieved by replicating the

highband, i.e. the high frequency part of the spectrum. A small amount of data representing a parametric description of the highband is encoded and used in the decoding process. The data rate is by far below the data rate required when using conventional AAC coding of the highband.

0.3.2.3 Parametric audio coding tools

The parametric audio coding tool **MPEG-4 HILN** (Harmonic and Individual Lines plus Noise) codes non-speech signals like music at bitrates of 4 kbit/s and higher using a parametric representation of the audio signal. The basic idea of this technique is to decompose the input signal into audio objects which are described by appropriate source models and represented by model parameters. Object models for sinusoids, harmonic tones, and noise are utilized in the HILN coder. HILN allows independent change of speed and pitch during decoding.

The Parametric Audio Coding tools combine very low bitrate coding of general audio signals with the possibility of modifying the playback speed or pitch during decoding without the need for an effects processing unit. In combination with the speech and audio coding tools in MPEG-4, improved overall coding efficiency is expected for applications of object based coding allowing selection and/or switching between different coding techniques.

This approach allows to introduce a more advanced source model than just assuming a stationary signal for the duration of a frame, which motivates the spectral decomposition used in e.g. the MPEG-4 General Audio Coder. As known from speech coding, where specialized source models based on the speech generation process in the human vocal tract are applied, advanced source models can be advantageous, especially for very low bitrate coding schemes.

Due to the very low target bitrates, only the parameters for a small number of objects can be transmitted. Therefore a perception model is employed to select those objects that are most important for the perceptual quality of the signal.

In HILN, the frequency and amplitude parameters are quantized according to the “just noticeable differences” known from psychoacoustics. The spectral envelope of the noise and the harmonic tones are described using LPC modeling as known from speech coding. Correlation between the parameters of one frame and those of consecutive frames is exploited by parameter prediction. Finally, the quantized parameters are entropy coded and multiplexed to form a bitstream payload.

A very interesting property of this parametric coding scheme arises from the fact that the signal is described in terms of frequency and amplitude parameters. This signal representation permits speed and pitch change functionality by simple parameter modification in the decoder. The HILN Parametric Audio Coder can be combined with MPEG-4 Parametric Speech Coder (HVXC) to form an integrated parametric coder covering a wider range of signals and bitrates. This integrated coder supports speed and pitch change. Using a speech/music classification tool in the encoder, it is possible to automatically select the HVXC for speech signals and the HILN for music signals. Such automatic HVXC/HILN switching was successfully demonstrated and the classification tool is described in the informative Annex of the MPEG-4 standard.

MPEG-4 SSC, (SinuSoidal Coding) is a parametric coding tool that is capable of full bandwidth high quality audio coding. The coding tool dissects a monaural or stereo audio signal into a number of different objects that each can be parameterized efficiently and encoded at a low bit-rate. These objects are, transients: representing dynamic changes in the temporal domain, sinusoids: representing deterministic components, and noise: representing components that do not have a clear temporal or spectral localisation. The fourth object, that is only relevant for stereo input signals, captures the stereo image. As the signal is represented in a parametric domain, independent, high quality pitch and tempo scaling are possible at low computational cost.

0.3.3 Lossless audio coding tools

MPEG-4 DST (Direct Stream Transfer) provides lossless coding of oversampled audio signals.

0.3.4 Synthesis tools

Synthesis tools are designed for very low bitrate description and transmission, and terminal-side synthesis, of synthetic music and other sounds.

The MPEG-4 toolset providing general audio synthesis capability is called **MPEG-4 Structured Audio**, and it is described in subpart 5 of ISO/IEC 14496-3. MPEG-4 Structured Audio (the SA coder) provides very general capabilities for the description of synthetic sound, and the normative creation of synthetic sound in the decoding terminal. High-quality stereo sound can be transmitted at bitrates from 0 kbit/s (no continuous cost) to 2-3 kbit/s for extremely expressive sound using these tools.

Rather than specify a particular method of synthesis, SA specifies a flexible language for describing methods of synthesis. This technique allows content authors two advantages. First, the set of synthesis techniques available is not limited to those that were envisioned as useful by the creators of the standard; any current or future method of synthesis may be used in MPEG-4 Structured Audio. Second, the creation of synthetic sound from structured descriptions is normative in MPEG-4, so sound created with the SA coder will sound the same on any terminal.

Synthetic audio is transmitted via a set of *instrument* modules that can create audio signals under the control of a *score*. An instrument is a small network of signal-processing primitives that control the parametric generation of sound according to some algorithm. Several different instruments may be transmitted and used in a single Structured Audio bitstream payload. A score is a time-sequenced set of commands that invokes various instruments at specific times to contribute their output to an overall music performance. The format for the description of instruments is SAOL, the Structured Audio Orchestra Language. The format for the description of scores is SASL, the Structured Audio Score Language.

Efficient transmission of sound samples, also called *wavetables*, for use in sampling synthesis is accomplished by providing interoperability with the MIDI Manufacturers Association Downloaded Sounds Level 2 (DLS-2) standard, which is normatively referenced by the Structured Audio standard. By using the DLS-2 format, the simple and popular technique of wavetable synthesis can be used in MPEG-4 Structured Audio soundtracks, either by itself or in conjunction with other kinds of synthesis using the more general-purpose tools. To further enable interoperability with existing content and authoring tools, the popular MIDI (Musical Instrument Digital Interface) control format can be used instead of, or in addition to, scores in SASL for controlling synthesis.

Through the inclusion of compatibility with MIDI standards, MPEG-4 Structured Audio thus represents a unification of the current technique for synthetic sound description (MIDI-based wavetable synthesis) with that of the future (general-purpose algorithmic synthesis). The resulting standard solves problems not only in very-low-bitrate coding, but also in virtual environments, video games, interactive music, karaoke systems, and many other applications.

0.3.5 Composition tools

Composition tools are designed for object-based coding, interactive functionality, and audiovisual synchronization.

The tools for audio composition, like those for visual composition, are specified in the MPEG-4 Systems standard (ISO/IEC 14496-1). However, since readers interested in audio functionality are likely to look here first, a brief overview is provided.

Audio composition is the use of multiple individual “audio objects” and mixing techniques to create a single soundtrack. It is analogous to the process of recording a soundtrack in a multichannel mix, with each musical instrument, voice actor, and sound effect on its own channel, and then “mixing down” the multiple channels to a single channel or single stereo pair. In MPEG-4, the multichannel mix itself may be transmitted, with each audio source using a different coding tool, and a set of instructions for mixdown also transmitted in the bitstream payload. As the multiple audio objects are received, they are decoded separately, but not played back to the listener; rather, the instructions for mixdown are used to prepare a single soundtrack from the “raw material” given in the objects. This final soundtrack is then played for the listener.

An example serves to illustrate the efficacy of this approach. Suppose, for a certain application, we wish to transmit the sound of a person speaking in a reverberant environment over stereo background music, at very high quality. A traditional approach to coding would demand the use of a general audio coding at 32 kbit/s/channel or above; the sound source is too complex to be well-modeled by a simple model-based coder. However, in MPEG-4 we can represent the soundtrack as the conjunction of several objects: a speaking person passed through a reverberator added to a synthetic music track. We transmit the speaker’s voice using the CELP tool at 16 kbit/s, the synthetic music using the SA tool at 2 kbit/s, and allow a small amount of overhead (only a few hundreds of bytes as a fixed cost) to describe the stereo mixdown and the reverberation. Using MPEG-4 and an object-based approach thus allows us to describe in less than 20 kbit/s total a stream that might require 64 kbit/s to transmit with traditional coding, at equivalent quality.

Additionally, having such structured soundtrack information present in the decoding terminal allows more sophisticated client-side interaction to be included. For example, the listener can be allowed (if the content author desires) to request that the background music be muted. This functionality would not be possible if the music and speech were coded into the same audio track.

With the **MPEG-4 Binary Format for Scenes (BIFS),** specified in MPEG-4 Systems, a subset tool called AudioBIFS allows content authors to describe sound scenes using this object-based framework. Multiple sources may be mixed and combined, and interactive control provided for their combination. Sample-resolution control over mixing is provided in this method. Dynamic download of custom signal-processing routines allows the content author to exactly request a particular, normative, digital filter, reverberator, or other effects-processing routine. Finally, an interface to terminal-dependent methods of 3-D audio spatialisation is provided for the description of virtual-reality and other 3-D sound material.

As AudioBIFS is part of the general BIFS specification, the same framework is used to synchronize audio and video, audio and computer graphics, or audio with other material. Please refer to ISO/IEC 14496-1 (MPEG-4 Systems) for more information on AudioBIFS and other topics in audiovisual synchronization.

0.3.6 Scalability tools

Scalability tools are designed for the creation of bitstream payloads that can be transmitted, without recoding, at several different bitrates.

Many of the stream types in MPEG-4 are *scalable* in one manner or another. Several types of scalability in the standard are discussed below.

Bitrate scalability allows a bitstream payload to be parsed into a bitstream payload of lower bitrate such that the combination can still be decoded into a meaningful signal. The bitstream payload parsing can occur either during transmission or in the decoder. Scalability is available within each of the natural audio coding schemes, or by a combination of different natural audio coding schemes.

Bandwidth scalability is a particular case of bitrate scalability, whereby part of a bitstream payload representing a part of the frequency spectrum can be discarded during transmission or decoding. This is available for the CELP speech coder, where an extension layer converts the narrow band base layer speech coder into a wide band speech coder. Also the general audio coding tools which all operate in the frequency domain offer a very flexible bandwidth control for the different coding layers.

Encoder complexity scalability allows encoders of different complexity to generate valid and meaningful bitstream payloads. An example for this is the availability of a high quality and a low complexity excitation module for the wideband CELP coder allowing to choose between significant lower encoder complexity or optimized coding quality.

Decoder complexity scalability allows a given bitstream payload to be decoded by decoders of different levels of complexity. A subtype of decoder complexity scalability is *graceful degradation*, in which a decoder dynamically monitors the resources available, and scales down the decoding complexity (and thus the audio quality) when resources are limited. The Structured Audio decoder allows this type of scalability; a content author may provide (for example) several different algorithms for the synthesis of piano sounds, and the content itself decides, depending on available resources, which one to use.

0.3.7 Upstream

Upstream tools are designed for the dynamic control the streaming of the server for bitrate control and quality feedback control.

The **MPEG-4 upstream** or backchannel allows a user on a remote side to dynamically control the streaming of MPEG-4 content from a server. Backchannel streams carrying the user interaction information.

0.3.8 Error robustness facilities

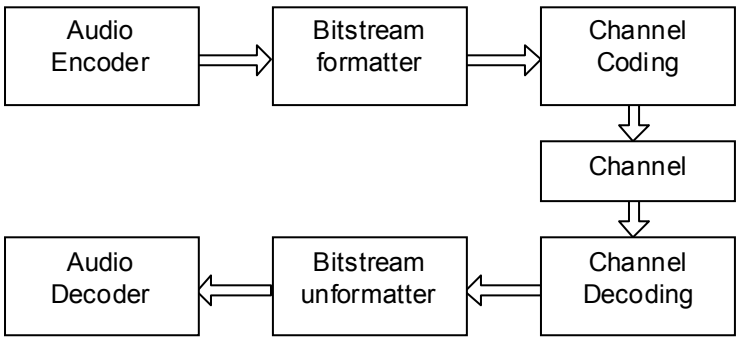
0.3.8.1 Overview

Error robustness facilities include tools for error resilience as well as for error protection.

The error robustness facilities provide improved performance on error-prone transmission channels. They are comprised of error resilient bitstream payload reordering, a common error protection tool and codec specific error resilience tools.

0.3.8.2 Error resilient bitstream payload reordering

Error resilient bitstream payload reordering allows the effective use of advanced channel coding techniques like unequal error protection (UEP), which can be perfectly adapted to the needs of the different coding tools. The basic idea is to rearrange the audio frame content depending on its error sensitivity in one or more instances belonging to different error sensitivity categories (ESC). This rearrangement can be either data element-wise or even bit-wise. An error resilient bitstream payload frame is build by concatenating these instances.



The basic principle is depicted in the figure above. A bitstream payload is reordered according to the error sensitivity of single bitstream payload elements or even single bits. This new arranged bitstream payload is channel coded, transmitted and channel decoded. Prior to audio decoding, the bitstream payload is rearranged to its original order.

0.3.8.3 Error protection

The EP tool provides unequal error protection. It receives several classes of bits from the audio coding tools, and then applies forward error correction codes (FEC) and/or cyclic redundancy codes (CRC) for each class, according to its error sensitivity.

The error protection tool (EP tool) provides the unequal error protection (UEP) capability to the set of ISO/IEC 14496-3 codecs. Main features of this tool are:

- providing a set of error correcting/detecting codes with wide and small-step scalability, both in performance and in redundancy
- providing a generic and bandwidth-efficient error protection framework, which covers both fixed-length frame streams and variable-length frame streams
- providing a UEP configuration control with low overhead.

0.3.8.4 Error resilience tools for AAC

Several tools are provided to increase the error resilience for AAC. These tools improve the perceived audio quality of the decoded audio signal in case of corrupted bitstream payloads, which may occur e. g. in the presence of noisy transmission channels.

- The *Virtual CodeBooks tool (VCB11)* extends the sectioning information of an AAC bitstream payload. This permits the detection of serious errors within the spectral data of an MPEG-4 AAC bitstream payload.

Virtual codebooks are used to limit the largest absolute value possible within any scalefactor band that uses escape values. While all virtual codeboocks use the codebook 11, the sixteen virtual codebooks introduced by VCB11 provide sixteen different limitations of the spectral values belonging to the corresponding subclause. Therefore, errors in the transmission of spectral data that result in spectral values exceeding the indicated limit can be located and appropriately concealed.

- The *Reversible Variable Length Coding tool (RVLC)* replaces the Huffman and DPCM coding of the scalefactors in an AAC bitstream payload. The RVLC uses symmetric codewords to enable both forward and backward decoding of the scalefactor data. In order to have a starting point for backward decoding, the total number of bits of the RVLC part of the bitstream payload is transmitted. Because of the DPCM coding of the scalefactors, also the value of the last scalefactor is transmitted to enable backward DPCM decoding. Since not all nodes of the RVLC code tree are used as codewords, some error detection is also possible.

- The *Huffman codeword reordering (HCR)* algorithm for AAC spectral data is based on the fact that some of the codewords can be placed at known positions so that these codewords can be decoded independent of any error within other codewords. Therefore, this algorithm avoids error propagation to those codewords, the so-called priority codewords (PCW). To achieve this, segments of known length are defined and those codewords are placed at the beginning of these segments. The remaining codewords (non-priority codewords, non-PCW) are filled into the gaps left by the PCWs using a special algorithm that minimizes error propagation to the non-PCWs codewords. This reordering algorithm does not increase the size of spectral data. Before applying the reordering algorithm, the PCWs are determined by sorting the codewords according to their importance.

Information technology — Coding of audio-visual objects —

Part 3:
Audio

ISO/IEC 14496-3 contains ten subparts:

Subpart 1: Main

Subpart 2: Speech coding — HVXC

Subpart 3: Speech coding — CELP

Subpart 4: General Audio coding (GA) — AAC, TwinVQ, BSAC

Subpart 5: Structured Audio (SA)

Subpart 6: Text To Speech Interface (TTSI)

Subpart 7: Parametric Audio Coding — HILN

Subpart 8: Parametric coding for high quality audio — SSC

Subpart 9: MPEG-1/2 Audio in MPEG-4

Subpart 10: Lossless coding of oversampled audio — DST

Content for Subpart 1

Subpart 1: Main

1.1 Scope

ISO/IEC 14496-3 (MPEG-4 Audio) is a new kind of audio International Standard that integrates many different types of audio coding: natural sound with synthetic sound, low bitrate delivery with high-quality delivery, speech with music, complex soundtracks with simple ones, and traditional content with interactive and virtual-reality content. By standardizing individually sophisticated coding tools as well as a novel, flexible framework for audio synchronization, mixing, and downloaded post-production, the developers of the MPEG-4 Audio standard have created new technology for a new, interactive world of digital audio.

MPEG-4, unlike previous audio standards created by ISO/IEC and other groups, does not target a single application such as real-time telephony or high-quality audio compression. Rather, MPEG-4 Audio is a standard that applies to *every* application requiring the use of advanced sound compression, synthesis, manipulation, or playback. The subparts that follow specify the state-of-the-art coding tools in several domains; however, MPEG-4 Audio is more than just the sum of its parts. As the tools described here are integrated with the rest of the MPEG-4 International Standard, exciting new possibilities for object-based audio coding, interactive presentation, dynamic soundtracks, and other sorts of new media, are enabled.

Since a single set of tools is used to cover the needs of a broad range of applications, *interoperability* is a natural feature of systems that depend on the MPEG-4 Audio International Standard. A system that uses a particular coder — for example, a real-time voice communication system making use of the MPEG-4 speech coding toolset — can easily share data and development tools with other systems, even in different domains, that use the same tool — for example, a voicemail indexing and retrieval system making use of MPEG-4 speech coding. A multimedia terminal that can decode the Natural Audio Profile of MPEG-4 Audio has audio capabilities that cover the entire spectrum of audio functionality available today and into the future.

1.2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 11172-3:1993, *Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s — Part 3: Audio*

ITU-T Rec.H.222.0(1995) | ISO/IEC 13818-1:2000, *Information technology — Generic coding of moving pictures and associated audio information — Part 1: Systems*

ISO/IEC 13818-3:1998, *Information technology — Generic coding of moving pictures and associated audio information — Part 3: Audio*

ISO/IEC 13818-7:2004, *Information technology – Generic coding of moving pictures and associated audio information — Part 7: Advanced Audio Coding (AAC)*

ISO/IEC 14496-1, *Information technology — Coding of audio-visual objects — Part 1: Systems*

ISO/IEC 14496-11, *Information technology — Coding of audio-visual objects — Part 11: Scene description and application engine*

ITU-T Recommendation H.223/Annex C: MULTIPLEXING PROTOCOL FOR LOW BITRATE MULTIMEDIA COMMUNICATION OVER HIGHLY ERROR_PRONE CHANNELS, April 1998.

(c) 1996 MIDI Manufacturers Association, *The Complete MIDI 1.0 Detailed Specification* v. 96.2

(c) 1998 MIDI Manufacturers Association, *The MIDI Downloadable Sounds Specification* v. 98.2

1.3 Terms and definitions

1.3.1. **AAC:** Advanced Audio Coding.

1.3.2. **AAC program:** A set of main audio channels, coupling channel, lfe channel and associated data streams intended to be decoded and played back simultaneously. A program may be defined by default, or specifically by a program_config_element(). A given single_channel_element(), channel_pair_element(), coupling_channel_element(), lfe_channel_element() or data_stream_element() may accompany one or more programs in any given stream.

1.3.3. **Audio access unit**: An individually accessible portion of audio data within an elementary stream.

1.3.4. **Audio composition unit**: An individually accessible portion of the output that an audio decoder produces from audio access units.

1.3.5. **Absolute time:** The time at which sound corresponding to a particular event is really created; time in the real-world. Contrast score time.

1.3.6. **Actual parameter:** The expression which, upon evaluation, is passed to an opcode as a parameter value.

1.3.7. **A-cycle:** See audio cycle.

1.3.8. **Adaptive codebook:** An approach to encode the long-term periodicity of the signal. The entries of the codebook consists of overlapping segments of past excitations.

1.3.9. **Alias:** Mirrored spectral component resulting from sampling.

1.3.10. **Analysis filterbank:** Filterbank in the encoder that transforms a broadband PCM audio signal into a set of spectral coefficients.

1.3.11. **Ancillary data:** Part of the bitstream payload that might be used for transmission of ancillary data.

1.3.12. **API:** Application Programming Interface.

1.3.13. **A-rate:** See audio rate.

1.3.14. **Asig:** The lexical tag indicating an a-rate variable.

1.3.15. **Audio buffer:** A buffer in the system target decoder (see ISO/IEC 13818-1) for storage of compressed audio data.

1.3.16. **Audio cycle:** The sequence of processing which computes new values for all a-rate expressions in a particular code block.

1.3.17. **Audio rate:** The rate type associated with a variable, expression or statement which may generate new values as often as the sampling rate.

1.3.18. **Audio sample:** A short snippet or clip of digitally represented sound. Typically used in wavetable synthesis.

1.3.19. **AudioBIFS:** The set of tools specified in ISO/IEC 14496-1 (MPEG-4 Systems) for the composition of audio data in interactive scenes.

1.3.20. **Authoring:** In Structured Audio, the combined processes of creatively composing music and sound control scripts, creating instruments which generate and alter sound, and encoding the instruments, control scripts, and audio samples in MPEG-4 Structured Audio format.

1.3.21. **Backus-Naur Format:** (BNF) A format for describing the syntax of programming languages, used here to specify the SAOL and SASL syntax.

1.3.22. **Backward compatibility:** A newer coding standard is backward compatible with an older coding standard if decoders designed to operate with the older coding standard are able to continue to operate by decoding all or part of a bitstream payload produced according to the newer coding standard.

1.3.23. **Bandwidth scalability:** The possibility to change the bandwidth of the signal during transmission.

1.3.24. **Bank:** A set of samples used together to define a particular sound or class of sounds with wavetable synthesis.

1.3.25. **Bark:** The Bark is the standard unit corresponding to one critical band width of human hearing.

1.3.26. **Beat:** The unit in which score time is measured.

1.3.27. **Bitrate:** The rate at which the compressed stream is delivered to the input of a decoder.

1.3.28. **Bitrate scalability:** The possibility to transmit a subset of the bitstream payload and still decode the bitstream payload with the same decoder.

1.3.29. **Bitstream payload verifier:** A process by which it is possible to test and verify that all the requirements specified in ISO/IEC 14496-3 are met by the bitstream payload.

1.3.30. **Bitstream; stream:** An ordered series of bits that forms the coded representation of the data.

1.3.31. **Block companding:** Normalizing of the digital representation of an audio signal within a certain time period.

1.3.32. **BNF:** See Backus-Naur Format.

1.3.33. **BSAC:** Bit Sliced Arithmetic Coding

1.3.34. **Bus:** An area in memory which is used to pass the output of one instrument into the input of another.

1.3.35. **Byte:** Sequence of 8-bits.

1.3.36. **Byte aligned:** A bit in a data function is byte-aligned if its position is a multiple of 8-bits from the first bit of this data function.

1.3.37. **CELP:** Code Excited Linear Prediction.

1.3.38. **Center channel:** An audio presentation channel used to stabilize the central component of the frontal stereo image.

1.3.39. **Channel:** A sequence of data representing an audio signal intended to be reproduced at one listening position.

1.3.40. **Coded audio bitstream:** A coded representation of an audio signal.

1.3.41. **Coded representation:** A data element as represented in its encoded form.

1.3.42. **Composition (compositing):** Using a scene description to mix and combine several separate audio tracks into a single presentation.

1.3.43. **Compression:** Reduction in the number of bits used to represent an item of data.

1.3.44. **Constant bitrate:** Operation where the bitrate is constant from start to finish of the coded bitstream.

1.3.45. **Context:** See state space.

1.3.46. **Control:** An instruction used to describe how to use a particular synthesis method to produce sound.

EXAMPLES:
“Using the piano instrument, play middle C at medium volume for 2 seconds.”
“Glissando the violin instrument up to middle C.”
“Turn off the reverberation for 8 seconds.”

1.3.47. **Control cycle:** The sequence of processing which computes new values for all control-rate expressions in a particular code block.

1.3.48. **Control period:** The length of time (typically measured in audio samples) corresponding to the control rate.

1.3.49. **Control rate:** (1) The rate at which instantiation and termination of instruments, parametric control of running instrument instances, sharing of global variables, and other non-sample-by-sample computation occurs in a particular orchestra. (2) The rate type of variables, expressions, and statements that can generate new values as often as the control rate.

1.3.50. **Core coder:** The term core coder is used to denote a base layer coder in certain scalability configurations. A core coder does not code the spectral samples of the MDCT filterbank of the subsequent AAC coding layers, but operates on a time domain signal. The output of the core decoder has to be up-sampled and transformed into the spectral domain, before it can be combined with the output of the AAC coding layers. Within the MPEG-4 Audio standard only the MPEG-4 CELP coder is a valid core coder. However, in principal, also another AAC coding layer, operating at a lower sampling rate, could be used on the time domain signal, and then combined with the other coding layer in exactly the same way as described for the CELP coder, and would therefore be called a core coder.

1.3.51. **CRC:** The Cyclic Redundancy Check to verify the correctness of data.

1.3.52. **Critical band:** This unit of bandwidth represents the standard unit of bandwidth expressed in human auditory terms, corresponding to a fixed length on the human cochlea. It is approximately equal to 100 Hz at low frequencies and 1/3 octave at higher frequencies, above approximately 700 Hz.

1.3.53. **Data element:** An item of data as represented after encoding and before decoding.

1.3.54. **Data function:** An encapsulation of data elements forming a logic unit.

1.3.55. **Decoded stream:** The decoded reconstruction of a compressed bitstream.

1.3.56. **Decoder:** An embodiment of a decoding process.

1.3.57. **Decoding (process):** The process that reads an input coded bitstream payload and outputs decoded audio samples.

1.3.58. **Demultiplexing:** Splitting one bitstream into several.

1.3.59. **DFT:** Discrete Fourier Transform.

1.3.60. **Digital storage media; DSM:** A digital storage or transmission device or system.

1.3.61. **Dimension conversion:** A method to convert a dimension of a vector by a combination of low pass filtering and linear interpolation.

1.3.62. **Discrete cosine transform; DCT:** Either the forward discrete cosine transform or the inverse discrete cosine transform. The DCT is an invertible, discrete orthogonal transformation.

1.3.63. **Downmix:** A matrixing of n channels to obtain less than n channels.

1.3.64. **DST:** Direct Stream Transfer

1.3.65. **Duration:** The amount of time between instantiation and termination of an instrument instance.

1.3.66. **Editing:** The process by which one or more coded bitstreams are manipulated to produce a new coded bitstream. Conforming edited bitstreams must meet the requirements defined in part 3 of ISO/IEC 14496.

1.3.67. **Elementary stream (ES):** A sequence of data that originates from a single producer in the transmitting MPEG-4 Terminal and terminates at a single recipient, e.g. an AVObject or a Control Entity in the receiving MPEG-4 Terminal. It flows through one FlexMux Channel.

1.3.68. **Encoder:** An embodiment of an encoding process.

1.3.69. **Encoding (process):** A process, not specified in ISO/IEC 14496, that reads a stream of input audio samples and produces a valid coded bitstream payload as defined in part 3 of ISO/IEC 14496.

1.3.70. **Enhancement layer(s):** The part(s) of the bitstream payload that is possible to drop in a transmission and still decode the bitstream payload.

1.3.71. **Entropy coding:** Variable length lossless coding of the digital representation of a signal to reduce statistical redundancy.

1.3.72. **Envelope:** A loudness-shaping function applied to a sound, or more generally, any function controlling a parametric aspect of a sound

1.3.73. **EP:** Error Protection

1.3.74. **ER:** Error resilience or Error Resilient (as appropriate)

1.3.75. **Event:** One control instruction.

1.3.76. **Excitation:** The excitation signal represents the input to the LPC module. The signal consists of contributions that cannot be covered by the LPC model.

1.3.77. **Expression:** A mathematical or functional combination of variable values, symbolic constants, and opcode calls.

1.3.78. **FFT:** Fast Fourier Transformation. A fast algorithm for performing a discrete Fourier transform (an orthogonal transform).

1.3.79. **Filterbank:** A set of band-pass filters covering the entire audio frequency range.

1.3.80. **Fine rate control:** The possibility to change the bitrate by, under some circumstances, skipping transmission of the LPC indices.

1.3.81. **Fixed codebook:** The fixed codebook contains excitation vectors for the speech synthesis filter. The contents of the codebook are non-adaptive (i.e. fixed).

1.3.82. **Flag:** A variable which can take one of only the two values defined in this specification.

1.3.83. **Formal parameter:** The syntactic element that gives a name to one of the parameters of an opcode.

1.3.84. **Forward compatibility:** A newer coding standard is forward compatible with an older coding standard if decoders designed to operate with the newer coding standard are able to decode bitstream payloads of the older coding standard.

1.3.85. **Frame**: A part of the audio signal that corresponds to a certain number of audio PCM samples.

1.3.86. **Fs:** Sampling frequency.

1.3.87. **FSS:** Frequency Selective Switch. Module which selects one of two input signals independently in each scalefactor band.

1.3.88. **Fundamental frequency:** A parameter which represents signal periodicity in frequency domain.

1.3.89. **Future wavetable:** A wavetable that is declared but not defined in the SAOL orchestra; its definition must arrive in the bitstream payload before it is used.

1.3.90. **Global block:** The section of the orchestra that describes global variables, route and send statements, sequence rules, and global parameters.

1.3.91. **Global context:** The state space used to hold values of global variables and wavetables.

1.3.92. **Global parameters:** The sampling rate, control rate, and number of input and output channels of audio associated with a particular orchestra.

1.3.93. **Global variable:** A variable that can be accessed and/or changed by several different instruments.

1.3.94. **Grammar:** A set of rules that describes the set of allowable sequences of lexical elements comprising a particular language.

1.3.95. **Guard expression:** The expression standing at the front of an if, while, or else statement that determines whether or how many times a particular block of code is executed.

1.3.96. **Hann window:** A time function applied sample-by-sample to a block of audio samples before Fourier transformation.

1.3.97. **Harmonic lines:** A set of spectral components having a common fundamental frequency.

1.3.98. **Harmonic magnitude:** Magnitude of each harmonic.

1.3.99. **Harmonic synthesis:** A method to obtain a periodic excitation from harmonic magnitudes.

1.3.100. **Harmonics:** Samples of frequency spectrum at multiples of the fundamental frequency.

1.3.101. **HCR:** Huffman codebook reordering

1.3.102. **HILN:** Harmonic and Individual Lines plus Noise (parametric audio coding).

1.3.103. **Huffman coding:** A specific method for entropy coding.

1.3.104. **HVXC:** Harmonic Vector eXcitation Coding (parametric speech coding).

1.3.105. **Hybrid filterbank:** A serial combination of subband filterbank and MDCT. Used in MPEG-1 and MPEG-2 Audio.

1.3.106. **I-cycle:** See initialisation cycle.

1.3.107. **IDCT:** Inverse Discrete Cosine Transform.

1.3.108. **Identifier:** A sequence of characters in a textual SAOL program that denotes a symbol.

1.3.109. **IFFT:** Inverse Fast Fourier Transform.

1.3.110. **IMDCT:** Inverse Modified Discrete Cosine Transform.

1.3.111. **Index:** Number indicating the quantized value(s).

1.3.112. **Individual line:** A spectral component described by frequency, amplitude and phase.

1.3.113. **Informative:** Aspects of a standards document that are provided to assist implementers, but are not required to be implemented in order for a particular system to be compliant to the standard.

1.3.114. **Initial phase:** A phase value at the onset of voiced signal in harmonic synthesis.

1.3.115. **Initialisation cycle:** See initialisation pass.

1.3.116. **Initialisation pass:** The sequence of processing that computes new values for each i-rate expression in a particular code block.

1.3.117. **Initialisation rate:** The rate type of variables, expressions, and statements that are set once at instrument instantiation and then do not change.

1.3.118. **Instance:** See instrument instantiation.

1.3.119. **Instantiation:** The process of creating a new instrument instantiation based on an event in the score or statement in the orchestra.

1.3.120. **Instrument:** An algorithm for parametric sound synthesis, described using SAOL. An instrument encapsulates all of the algorithms needed for one sound-generation element to be controlled with a score.

NOTE - An MPEG-4 Structured Audio instrument does not necessarily correspond to a real-world instrument. A single instrument might be used to represent an entire violin section, or an ambient sound such as the wind. On the other hand, a single real-world instrument that produces many different timbres over its performance range might be represented using several SAOL instruments.

1.3.121. **Instrument instantiation:** The state space created as the result of executing a note-creation event with respect to a SAOL orchestra.

1.3.122. **Intensity stereo:** A method of exploiting stereo irrelevance or redundancy in stereophonic audio programmes based on retaining at high frequencies only the energy envelope of the right and left channels.

1.3.123. **Interframe prediction:** A method to predict a value in the current frame from values in the previous frames. Interframe prediction is used in VQ of LSP.

1.3.124. **International Phonetic Alphabet; IPA :** The worldwide agreed symbol set to represent various phonemes appearing in human speech.

1.3.125. **I-pass:** See initialisation pass.

1.3.126. **IPQF:** inverse polyphase quadrature filter

1.3.127. **I-rate:** See initialisation rate.

1.3.128. **Ivar:** The lexical tag indicating an i-rate variable.

1.3.129. **Joint stereo coding:** Any method that exploits stereophonic irrelevance or stereophonic redundancy.

1.3.130. **Joint stereo mode:** A mode of the audio coding algorithm using joint stereo coding.

1.3.131. **K-cycle:** See control cycle.

1.3.132. **K-rate:** See control rate.

1.3.133. **Ksig:** The lexical tag indicating a k-rate variable.

1.3.134. **Lexical element:** See token.

1.3.135. **Lip shape pattern :** A number that specifies a particular pattern of the preclassified lip shape.

1.3.136. **Lip synchronization :** A functionality that synchronizes speech with corresponding lip shapes.

1.3.137. **Looping:** A typical method of wavetable synthesis. Loop points in an audio sample are located and the sound between those endpoints is played repeatedly while being simultaneously modified by envelopes, modulators, etc.

1.3.138. **Low frequency enhancement (LFE) channel:** A limited bandwidth channel for low frequency audio effects in a multichannel system.

1.3.139. **LPC:** Linear Predictive Coding.

1.3.140. **LPC residual signal:** A signal filtered by the LPC inverse filter, which has a flattened frequency spectrum.

1.3.141. **LPC synthesis filter:** An IIR filter whose coefficients are LPC coefficients. This filter models the time varying vocal tract.

1.3.142. **LSP:** Line Spectral Pairs.

1.3.143. **LTP:** Long Term Prediction.

1.3.144. **M/S stereo:** A method of removing imaging artifacts as well as exploiting stereo irrelevance or redundancy in stereophonic audio programs based on coding the sum and difference signal instead of the left and right channels.

1.3.145. **Main audio channels:** All single_channel_elements or channel_pair_elements in one program.

1.3.146. **Mapping:** Conversion of an audio signal from time to frequency domain by subband filtering and/or by MDCT.

1.3.147. **Masking:** A property of the human auditory system by which an audio signal cannot be perceived in the presence of another audio signal.

1.3.148. **Masking threshold:** A function in frequency and time below which an audio signal cannot be perceived by the human auditory system.

1.3.149. **MIDI:** The Musical Instrument Digital Interface standards. Certain aspects of the MPEG-4 Structured Audio tools provide interoperability with MIDI standards.

1.3.150. **Mixed voiced frame:** A speech segment which has both voiced and unvoiced components.

1.3.151. **Modified discrete cosine transform (MDCT):** A transform which has the property of time domain aliasing cancellation.

1.3.152. **Moving picture dubbing :** A functionality that assigns synthetic speech to the corresponding moving picture while utilizing lip shape pattern information for synchronization.

1.3.153. **MPE:** Multi Pulse Excitation.

1.3.154. **MPEG-4 Audio Text-to-Speech Decoder :** A device that produces synthesized speech by utilizing the M-TTS bitstream payload while supporting all the M-TTS functionalities such as speech synthesis for FA and MP dubbing.

1.3.155. **M-TTS sentence :** This defines the information such as prosody, gender, and age for only the corresponding sentence to be synthesized.

1.3.156. **M-TTS sequence :** This defines the control information which affects all M-TTS sentences that follow this M-TTS sequence.

1.3.157. **Multichannel:** A combination of audio channels used to create a spatial sound field.

1.3.158. **Multilingual:** A presentation of dialogue in more than one language.

1.3.159. **Multiplexing:** Combining several bitstream payloads into one.

1.3.160. **Natural Sound:** A sound created through recording from a real acoustic space. Contrasted with synthetic sound.

1.3.161. **NCC:** Number of Considered Channels. In case of AAC, it is the number of channels represented by the elements SCE, independently switched CCE and CPE, i.e. once the number of SCEs plus once the number of independently switched CCEs plus twice the number of CPEs. With respect to the naming conventions of the MPEG-AAC decoders and payloads, NCC=A+I. This number is used to derive the required decoder input buffer size (see subpart 4, subclause 4.5.3.1). In case of other codecs, it tis the total number of channels.

1.3.162. **Noise component:** A signal component modeled as noise.

1.3.163. **Non-tonal component:** A noise-like component of an audio signal.

1.3.164. **Normative:** Those aspects of a standard that must be implemented in order for a particular system to be compliant to the standard.

1.3.165. **Nyquist sampling:** Sampling at or above twice the maximum bandwidth of a signal.

1.3.166. **OD:** Object Descriptor.

1.3.167. **Opcode:** A parametric signal-processing function that encapsulates a certain functionality so that it may be used by several instruments.

1.3.168. **Orchestra:** The set of sound-generation and sound-processing algorithms included in an MPEG-4 bitstream payload. Includes instruments, opcodes, routing, and global parameters.

1.3.169. **Orchestra cycle:** A complete pass through the orchestra, during which new instrument instantiations are created, expired ones are terminated, each instance receives one k-cycle and one control period worth of a-cycles, and output is produced.

1.3.170. **Padding:** A method to adjust the average length of an audio frame in time to the duration of the corresponding PCM samples, by conditionally adding a slot to the audio frame.

1.3.171. **Parameter:** A variable within the syntax of this specification which may take one of a range of values. A variable which can take one of only two values is a flag or indicator and not a parameter.

1.3.172. **Parameter fields:** The names given to the parameters to an instrument.

1.3.173. **Parser:** Functional stage of a decoder which extracts from a coded bitstream payload a series of bits representing coded elements.

1.3.174. **P-fields:** See parameter fields.

1.3.175. **Phoneme/bookmark-to-FAP converter :** A device that converts phoneme and bookmark information to FAPs.

1.3.176. **Pi:** The constant π = 3.14159...

1.3.177. **Pitch:** A parameter which represents signal periodicity in the time domain. It is expressed in terms of the number of samples.

1.3.178. **Pitch control:** A functionality to control the pitch of the synthesized speech signal without changing its speed.

1.3.179. **PNS:** Perceptual Noise Substitution.

1.3.180. **Polyphase filterbank:** A set of equal bandwidth filters with special phase interrelationships, allowing an efficient implementation of the filterbank.

1.3.181. **Postfilter:** A filter to enhance the perceptual quality of the synthesized speech signal.

1.3.182. **PQF:** polyphase quadrature filter

1.3.183. **Prediction:** The use of a predictor to provide an estimate of the sample value or data element currently being decoded.

1.3.184. **Prediction error:** The difference between the actual value of a sample or data element and its predictor.

1.3.185. **Predictor:** A linear combination of previously decoded sample values or data elements.

1.3.186. **Presentation channel:** An audio channel at the output of the decoder.

1.3.187. **Production rule:** In Backus-Naur Form grammars, a rule that describes how one syntactic element may be expressed in terms of other lexical and syntactic elements.

1.3.188. **PSNR:** Peak Signal to Noise Ratio.

1.3.189. **Psychoacoustic model:** A mathematical model of the masking behaviour of the human auditory system.

1.3.190. **Random access:** The process of beginning to read and decode the coded stream at an arbitrary point.

1.3.191. **Rate semantics:** The set of rules describing how rate types are assigned to variables, expressions, statements, and opcodes, and the normative restrictions that apply to a bitstream regarding combining these elements based on their rate types.

1.3.192. **Rate type:** The “speed of execution” associated with a particular variable, expression, statement, or opcode.

1.3.193. **Rate-mismatch error:** The condition that results when the rate semantics rules are violated in a particular SAOL construction. A type of syntax error.

1.3.194. **Reserved:** The term "reserved" when used in the subclauses defining the coded bitstream payload indicates that the value may be used in the future for ISO/IEC defined extensions.

1.3.195. **Route statement:** A statement in the global block that describes how to place the output of a certain set of instruments onto a bus.

1.3.196. **RPE:** Regular Pulse Excitation.

1.3.197. **Run-time error:** The condition that results from improper calculations or memory accesses during execution of a SAOL orchestra.

1.3.198. **RVLC:** Reversible Variable Length Coding

1.3.199. **Sample:** See Audio sample.

1.3.200. **Sample Bank Format:** A component format of MPEG-4 Structured Audio that allows the description of a set of samples for use in wavetable synthesis and processing methods to apply to them.

1.3.201. **Sampling Frequency (Fs):** Defines the rate in Hertz which is used to digitize an audio signal during the sampling process.

1.3.202. **SAOL:** The Structured Audio Orchestra Language, pronounced like the English word “sail”. SAOL is a digital-signal processing language that allows for the description of arbitrary synthesis and control algorithms as part of the content bitstream payload.

1.3.203. **SAOL orchestra:** See orchestra.

1.3.204. **SASBF:** The MPEG-4 Structured Audio Sample Bank Format, an efficient format for the transmission of blocks of wavetable (sample data) compatible with the MIDI method for the same.

1.3.205. **SASL:** The Structured Audio Score Language. SASL is a simple format that allows for powerful and flexible control of music and sound synthesis.

1.3.206. **SBA:** Segmented Binary Arithmetic Coding which is the error resilient tool for BSAC.

1.3.207. **SBR**: Spectral Band Replication.

1.3.208. **Scalefactor:** Factor by which a set of values is scaled before quantization.

1.3.209. **Scalefactor band:** A set of spectral coefficients which are scaled by one scalefactor.

1.3.210. **Scalefactor index:** A numerical code for a scalefactor.

1.3.211. **Scheduler:** The component of MPEG-4 Structured Audio that describes the mapping from control instructions to sound synthesis using the specified synthesis techniques. The scheduler description provides normative bounds on event-dispatch times and responses.

1.3.212. **Scope :** The code within which access to a particular variable name is allowed.

1.3.213. **Score:** A description in some format of the sequence of control parameters needed to generate a desired music composition or sound scene. In MPEG-4 Structured Audio, scores are described in SASL and/or MIDI.

1.3.214. **Score time:** The time at which an event happens in the score, measured in beats. Score time is mapped to absolute time by the current tempo.

1.3.215. **Semantics:** The rules describing what a particular instruction or bitstream payload element should do. Most aspects of bitstream payload and SAOL semantics are normative in MPEG-4.

1.3.216. **Send statement:** A statement in the global block that describes how to pass a bus on to an effect instrument for post-processing.

1.3.217. **Sequence rules:** The set of rules, both default and explicit, given in the global block that define in what order to execute instrument instantiations during an orchestra cycle.

1.3.218. **SIAQ**: Scalable Inverse AAC Quantization Module.

1.3.219. **Side information:** Information in the bitstream payload necessary for controlling the decoder.

1.3.220. **Signal variable:** A unit of memory, labelled with a name, that holds intermediate processing results. Each signal variable in MPEG-4 Structured Audio is instantaneously representable by a 32-bit floating point value.

1.3.221. **Sinusoidal synthesis:** A method to obtain a time domain waveform by a sum of amplitude modulated sinusoidal waveforms.

1.3.222. **Spatialisation:** The process of creating special sounds that a listener perceives as emanating from a particular direction.

1.3.223. **Spectral coefficients:** Discrete frequency domain data output from the analysis filterbank.

1.3.224. **Spectral envelope:** A set of harmonic magnitudes.

1.3.225. **Speed control:** A functionality to control the speed of the synthesized speech signal without changing its pitch or phonemes.

1.3.226. **Spreading function:** A function that describes the frequency spread of masking effects.

1.3.227. **SQ:** Scalar Quantization.

1.3.228. **SSC**: SinuSoidal Coding, parametric coder for high quality audio.

1.3.229. **State space:** A set of variable-value associations that define the current computational state of an instrument instantiation or opcode call. All the “current values” of the variables in an instrument or opcode call.

1.3.230. **Statement:** “One line” of a SAOL orchestra.

1.3.231. **Stereo-irrelevant:** A portion of a stereophonic audio signal which does not contribute to spatial perception.

1.3.232. **Structured audio:** Sound-description methods that make use of high-level models of sound generation and control. Typically involving synthesis description, structured audio techniques allow for ultra-low bitrate description of complex, high-quality sounds.

1.3.233. **Stuffing (bits); stuffing (bytes):** Code-words that may be inserted at particular locations in the coded bitstream payload that are discarded in the decoding process. Their purpose is to increase the bitrate of the stream which would otherwise be lower than the desired bitrate.

1.3.234. **Surround channel:** An audio presentation channel added to the front channels (L and R or L, R, and C) to enhance the spatial perception.

1.3.235. **Symbol:** A sequence of characters in a SAOL program, or a symbol token in a MPEG-4 Structured Audio bitstream payload, that represents a variable name, instrument name, opcode name, table name, bus name, etc.

1.3.236. **Symbol table:** In an MPEG-4 Structured Audio bitstream payload, a sequence of data that allows the tokenised representation of SAOL and SASL code to be converted back to a readable textual representation. The symbol table is an optional component.

1.3.237. **Symbolic constant:** A floating-point value explicitly represented as a sequence of characters in a textual SAOL orchestra, or as a token in a bitstream payload.

1.3.238. **Syncword:** A code embedded in audio transport streams that identifies the start of a transport frame.

1.3.239. **Syntax:** The rules describing what a particular instruction or bitstream payload element should look like. All aspects of bitstream payload and SAOL syntax are normative in MPEG-4.

1.3.240. **Syntax error:** The condition that results when a bitstream payload element does not comply with its governing rules of syntax.

1.3.241. **Synthesis:** The process of creating sound based on algorithmic descriptions.

1.3.242. **Synthesis filterbank:** Filterbank in the decoder that reconstructs a PCM audio signal from subband samples.

1.3.243. **Synthetic Sound:** Sound created through synthesis.

1.3.244. **Tempo:** The scaling parameter that specifies the relationship between score time and absolute time. A tempo of 60 beats per minute means that the score time measured in beats is equivalent to the absolute time measured in seconds; higher numbers correspond to faster tempi, so that 120 beats per minute is twice as fast.

1.3.245. **Terminal:** The “client side” of an MPEG transaction; whatever hardware and software are necessary in a particular implementation to allow the capabilities described in this document.

1.3.246. **Termination:** The process of destroying an instrument instantiation when it is no longer needed.

1.3.247. **Text-to-speech synthesizer :** A device producing synthesized speech according to the input sentence character strings.

1.3.248. **Timbre:** The combined features of a sound that allow a listener to recognise such aspects as the type of instrument, manner of performance, manner of sound generation, etc. Those aspects of sound that distinguish sounds equivalent in pitch and loudness.

1.3.249. **TNS:** Temporal Noise Shaping

1.3.250. **Token:** A lexical element of a SAOL orchestra a keyword, punctuation mark, symbol name, or symbolic constant.

1.3.251. **Tokenisation:** The process of converting a orchestra in textual SAOL format into a bitstream payload representation consisting of a stream of tokens.

1.3.252. **Tonal component:** A sinusoid-like component of an audio signal.

1.3.253. **Trick mode :** A set of functions that enables stop, play, forward, and backward operations for users.

1.3.254. **TTSI:** Text to Speech Interface.

1.3.255. **TwinVQ:** Transform domain Weighted Interleave Vector Quantization.

1.3.256. **Unvoiced frame:** Frame containing unvoiced speech which looks like random noise with no periodicity.

1.3.257. **V/UV decision:** Decision whether the current frame is voiced or unvoiced or mixed voiced.

1.3.258. **Variable:** See signal variable.

1.3.259. **Variable bitrate:** Operation where the bitrate varies with time during the decoding of a coded stream.

1.3.260. **Variable length code (VLC):** A code word assigned by variable length encoder (See variable length coding).

1.3.261. **Variable length coding:** A reversible procedure for coding that assigns shorter code-words to frequent symbols and longer code-words to less frequent symbols.

1.3.262. **Variable length decoder:** A procedure to obtain the symbols encoded with a variable length coding technique.

1.3.263. **Variable length encoder:** A procedure to assign variable length codewords to symbols.

1.3.264. **VCB11:** Virtual Codebooks for codebook 11.

1.3.265. **Vector quantizer:** Tool that quantizes several values to one index.

1.3.266. **Virtual codebook:** If several codebook values refer to one and the same physical codebook, these values are called virtual codebooks.

1.3.267. **Voiced frame:** A voiced speech segment is known by its relatively high energy content, but more importantly it contains periodicity which is called the pitch of the voiced speech.

1.3.268. **VQ:** Vector Quantization.

1.3.269. **VXC:** Vector eXcitation Coding. It is also called CELP (Coded Excitation Linear Prediction). In HVXC, no adaptive codebook is used.

1.3.270. **Wavetable synthesis:** A synthesis method in which sound is created by simple manipulation of audio samples, such as looping, pitch-shifting, enveloping, etc.

1.3.271. **White Gaussian noise:** A noise sequence which has a Gaussian distribution.

1.3.272. **Width:** The number of channels of data that an expression represents.

1.4 Symbols and abbreviations

The mathematical operators used in this part of ISO/IEC 14496 are similar to those used in the C programming language. However, integer division with truncation and rounding are specifically defined. The bitwise operators are defined assuming two's-complement representation of integers. Numbering and counting loops generally begin from zero.

1.4.1 Arithmetic operators

+ Addition.

– Subtraction (as a binary operator) or negation (as a unary operator).

++ Increment.

– – Decrement.

* Multiplication.

^ Power.

/ Integer division with truncation of the result toward zero. For example, 7/4 and –7/–4 are truncated to 1 and –7/4 and 7/–4 are truncated to –1.

// Integer division with rounding to the nearest integer. Half-integer values are rounded away from zero unless otherwise specified. For example 3//2 is rounded to 2, and –3//2 is rounded to –2.

DIV Integer division with truncation of the result towards $-\infty$.

| | Absolute value. | x | = x when x > 0
| x | = 0 when x == 0
| x | = –x when x < 0

% Modulus operator. Defined only for positive numbers.

Sign( ) Sign. Sign(x) = 1 when x > 0
Sign(x) = 0 when x == 0
Sign(x) = −1 when x < 0

INT ( ) Truncation to integer operator. Returns the integer part of the real-valued argument.

NINT ( ) Nearest integer operator. Returns the nearest integer value to the real-valued argument. Half-integer values are rounded away from zero.

sin Sine.

cos Cosine.

exp Exponential.

√ Square root.

$\log_{10}$ Logarithm to base ten.

$\log_e$ Logarithm to base e.

$\log_2$ Logarithm to base 2.

ceil( ) Ceiling operator. Returns the smallest integer that is greater than or equal to the real-valued argument.

1.4.2 Logical operators

|| Logical OR.

&& Logical AND.

! Logical NOT

1.4.3 Relational operators

> Greater than.

>= Greater than or equal to.

< Less than.

<= Less than or equal to.

== Equal to.

!= Not equal to.

max [,...,] the maximum value in the argument list.

min [,...,] the minimum value in the argument list.

1.4.4 Bitwise operators

A twos complement number representation is assumed where the bitwise operators are used.

& AND

|	OR
>>	Shift right with sign extension.
<<	Shift left with zero fill.

1.4.5 Assignment

=	Assignment operator.

1.4.6 Mnemonics

The following mnemonics are defined to describe the different data types used in the coded bitstream payload.

bslbf	Bit string, left bit first, where "left" is the order in which bit strings are written in ISO/IEC 14496. Bit strings are written as a string of 1s and 0s within single quote marks, e.g. '1000 0001'. Blanks within a bit string are for ease of reading and have no significance.
L, C, R, LS, RS	Left, center, right, left surround and right surround audio signals
rpchof	Remainder polynomial coefficients, highest order first. (Audio)
uimsbf	Unsigned integer, most significant bit first.
vlclbf	Variable length code, left bit first, where "left" refers to the order in which the VLC codes are written.
window	Number of the actual time slot in case of block_type == 2, 0 <= window <= 2. (Audio)

The byte order of multi-byte words is most significant byte first.

1.4.7 Constants

π	3,14159265358...
e	2,71828182845...

1.4.8 Method of describing bitstream payload syntax

The bitstream payload retrieved by the decoder is described in the syntax section of each subpart. Each data item in the bitstream payload is in bold type.

It is described by

- its name
- its length in bits, where "X..Y" indicates that the number of bits is one of the values between X and Y including X and Y. "{X;Y}" means the number of bits is X or Y, depending on the value of other data elements in the bitstream payload.),
- a mnemonic for its type and order of transmission.

Data elements forming a logic unit are encapsulated in data functions.

data_function ( );	Data function call.

data_function ( ) {
. . .
}
	Data function entity.

The action caused by a decoded data element in a bitstream payload depends on the value of that data element and on data elements previously decoded. The decoding of the data elements and the definition of the state variables used in their decoding are described in in the subclauses following the syntax section of each subpart. The following constructs are used to express the conditions when data elements are present, and are in normal type:

Note this syntax uses the 'C'-code convention that a variable or expression evaluating to a non-zero value is equivalent to a condition that is true.

while (condition) {
data_element
. . .
}
	If the condition is true, then the group of data elements occurs next in the data stream. This repeats until the condition is not true.

do { **data_element** . . . } while (condition)	The data element always occurs at least once. The data element is repeated until the condition is not true.

if (condition) {
data_element
. . .
}
else {
data_element
. . .
}
	If the condition is true, then the first group of data elements occurs next in the data stream
	
	If the condition is not true, then the second group of data elements occurs next in the data stream.

switch (expression) {
case const-expr:
data_element;
break;
case const-expr:
data_element;
}
	If the condition formed by the comparison of expression and const-expr is true, then the data stream continues with the subsequent data elements. An optionally break statement can be used to immediately leave the switch, data elements beyond a break do not occur in the data stream.

for (expr1; expr2; expr3)
{
data_element
. . .
}
	Expr1 is an expression specifying the initialisation of the loop. Normally it specifies the initial state of the counter. Expr2 is a condition specifying a test made before each iteration of the loop. The loop terminates when the condition is not true. Expr3 is an expression that is performed at the end of each iteration of the loop, normally it increments a counter.

Note that the most common usage of this construct is as follows:

for (i = 0; i < n; i++) {
data_element
. . .
}
	The group of data elements occurs n times. Conditional constructs within the group of data elements may depend on the value of the loop control variable i, which is set to zero for the first occurrence, incremented to one for the second occurrence, and so forth.

As noted, the group of data elements may contain nested conditional constructs. For compactness, the {} may be omitted when only one data element follows.

data_element [ ]	data_element [ ] is an array of data. The number of data elements is indicated by the context.
data_element [n]	data_element [n] is the n+1th element of an array of data.
data_element [m][n]	data_element [m][n] is the m+1,n+1 th element of a two-dimensional array of data.
data_element [l][m][n]	data_element [l][m][n] is the l+1,m+1,n+1 th element of a three-dimensional array of data.
data_element [m..n]	data_element [m..n]is the inclusive range of bits between bit m and bit n in the data_element.

While the syntax is expressed in procedural terms, it should not be assumed that this implements a satisfactory decoding procedure. In particular, it defines a correct and error-free input bitstream payload. Actual decoders must include a means to deal with incorrect bitstream payloads and to find the start of the described elements.

Definition of nextbits function

The function nextbits() permits comparison of a bit string with the next bits to be decoded in a stream.

1.5 Technical overview

1.5.1 MPEG-4 audio object types

1.5.1.1 Audio object type definition

Table 1.1 — Audio Object Type definition based on Tools/Modules

Object Type ID	Audio Object Type	gain control	block switching	window shapes - standard	window shapes – AAC LD	filterbank - standard	filterbank - SSR	TNS	LTP	intensity	coupling	frequency deomain prediction	PNS	MS	SIAQ	FSS	upsampling filter tool	quantisation&coding - AAC	quantisation&coding – TwinVQ	quantisation&coding - BSAC	AAC ER Tools	ER payload syntax	EP Tool 1)	CELP	Silence Compression	HVXC	HVXC 4kbit/s VR	SA tools	SASBF	MIDI	HILN	TTSI	SBR	Layer-1	Layer-2	Layer-3	SSC (Transient, Sinusoid, Noise)	Parametric stereo	DST	Remark
0	Null																																							
1	AAC main		X	X		X		X		X	X	X	X	X				X																						2)
2	AAC LC		X	X		X		X		X	X		X	X				X																						
3	AAC SSR	X	X	X			X	X		X	X		X	X				X																						
4	AAC LTP		X	X		X		X	X	X	X		X	X				X																						2)
5	SBR																																X							6)
6	AAC Scalable		X	X		X		X	X	X			X	X	X	X	X	X																						

Object Type ID	Audio Object Type	gain control	block switching	window shapes - standard	window shapes – AAC LD	filterbank - standard	filterbank - SSR	TNS	LTP	intensity	coupling	frequency deomain prediction	PNS	MS	SIAQ	FSS	upsampling filter tool	quantisation&coding - AAC	quantisation&coding – TwinVQ	quantisation&coding - BSAC	AAC ER Tools	ER payload syntax	EP Tool 1)	CELP	Silence Compression	HVXC	HVXC 4kbit/s VR	SA tools	SASBF	MIDI	HILN	TTSI	SBR	Layer-1	Layer-2	Layer-3	SSC (Transient, Sinusoid, Noise)	Parametric stereo	DST	Remark
7	TwinVQ		X	X		X		X	X					X					X																					
8	CELP																							X																
9	HVXC																									X														
10	(reserved)																																							
11	(reserved)																																							
12	TTSI																															X								
13	Main synthetic																											X	X	X										3)
14	Wavetable synthesis																												X	X										4)
15	General MIDI																													X										
16	Algorithmic Synthesis and Audio FX																											X												
17	ER AAC LC		X	X		X		X		X			X	X				X			X	X	X																
18	(reserved)																																							
19	ER AAC LTP		X	X		X		X	X	X			X	X				X			X	X	X																5)
20	ER AAC scalable		X	X		X		X		X			X	X	X	X	X	X			X	X	X																6)
21	ER TwinVQ		X	X		X		X						X					X			X	X																
22	ER BSAC		X	X		X		X		X			X	X						X		X	X																
23	ER AAC LD				X	X		X	X	X			X	X				X			X	X	X																
24	ER CELP																					X	X	X	X														
25	ER HVXC																					X	X			X	X												
26	ER HILN																					X	X								X								
27	ER Parametric																					X	X			X	X				X								
28	SSC																																			X	X		
29	(reserved)																																							
30	(reserved)																																							
31	(escape)																																							
32	Layer-1																																	X						
33	Layer-2																																		X					
34	Layer-3																																			X				
35	DST																																					X		
36 - 95	(reserved)																																							

Notes:

1) The bit parsing function is mandatory on decoder site. However, the error detection and error correction functions are optional.

2) Contains AAC LC.

3) Contains Wavetable synthesis and Algorithmic Synthesis and Audio FX.

4) Contains General MIDI.

5) Contains ER AAC LC.

6) The upsampling filter tool is only required in combination with a core coder.

1.5.1.2 Description

1.5.1.2.1 NULL object type

The NULL object provides the possibility to feed raw PCM data directly to the audio compositor. No decoding is involved, However, an audio object descriptor is used to specify the sampling rate and the audio channel configuration.

1.5.1.2.2 AAC - Main object type

The AAC Main object is very similar to the AAC Main Profile that is defined in ISO/IEC 13818-7. However, additionally the PNS tool is available. The restrictions of the AAC Main profile with respect to multiple programs and mixdown elements also apply to the AAC Main object type. The AAC Main object type bitstream payload syntax is compatible with the syntax defined in ISO/IEC 13818-7. All the MPEG-2 AAC multi-channel capabilities are available. A decoder capable to decode a MPEG-4 main object stream can also parse and decode a MPEG-2 AAC raw data stream. On the other hand, although a MPEG-2 AAC coder can parse an MPEG-4 AAC Main bitstream payload, decoding may fail, since PNS might have been used.

1.5.1.2.3 AAC - Low Complexity (LC) object type

The MPEG-4 AAC Low Complexity object type is the counterpart to the MPEG-2 AAC Low Complexity Profile, with exactly the same restrictions as mentioned above for the AAC Main object type.

1.5.1.2.4 AAC - Scalable Sampling Rate (SSR) object type

The MPEG-4 AAC Scalable Sampling Rate object type is the counterpart to the MPEG-2 AAC Scalable Sampling Rate Profile, with exactly the same restrictions as mentioned above for the AAC Main object type.

1.5.1.2.5 AAC - Long Term Predictor (LTP) object type

The MPEG-4 AAC LTP object type is similar to the AAC Main object type. However, a Long Term Predictor replaces the MPEG-2 AAC predictor. The LTP achieves a similar coding gain, but requires significantly lower implementation complexity. The bitstream payload syntax for this object type is very similar to the syntax defined in ISO/IEC 13818-7. An MPEG-2 AAC LC profile bitstream payload can be decoded without restrictions using an MPEG-4 AAC LTP object decoder.

1.5.1.2.6 SBR object type

The SBR Object contains the SBR-Tool and can be combined with the audio object types indicated in Table 1.2.

Table 1.2 — Audio object types that can be combined with the SBR Tool

Audio Object Type	Combination with SBR Tool permitted	Object Type ID
Null		0
AAC main	X	1
AAC LC	X	2
AAC SSR	X	3
AAC LTP	X	4
SBR		5
AAC Scalable	X	6
TwinVQ		7
CELP		8
HVXC		9
(Reserved)		10
(Reserved)		11
TTSI		12
Main synthetic		13
Wavetable synthesis		14
General MIDI		15
Algorithmic Synthesis and Audio FX		16
ER AAC LC	X	17
(Reserved)		18
ER AAC LTP	X	19
ER AAC scalable	X	20
ER TwinVQ		21
ER BSAC		22
ER AAC LD		23
ER CELP		24
ER HVXC		25
ER HILN		26
ER Parametric		27
SSC		28
(Reserved)		29
(Reserved)		30
(Reserved)		31

1.5.1.2.7 AAC Scalable object type

The scalable AAC object uses a different bitstream payload syntax to support bitrate- and bandwidth- scalability. A large number of scalable combinations are available, including combinations with TwinVQ and CELP coder tools. However, only mono, or 2-channel stereo objects are supported.

1.5.1.2.8 TwinVQ object type

The TwinVQ object belongs to the GA coding scheme that quantizes the MDCT coefficients. This coding scheme is based on fixed rate vector quantization instead of Huffman coding in AAC.

Low bit rate mono and stereo audio coding is available. Scalable audio coding schemes are also available in the Scalable Audio Profile combined with AAC scalable object type.

1.5.1.2.9 CELP object type

The CELP object is supported by the CELP speech coding tools, which provide coding at 8 kHz and 16 kHz sampling rate at bit rates in the range 4-24 kbit/s. Additionally, bit rate scalability and bandwidth scalability are

available in order to provide scalable decoding of CELP streams. CELP Object always contains exactly one mono audio signal.

1.5.1.2.10 HVXC object type

The HVXC object is supported by the parametric speech coding (HVXC) tools, which provide fixed bitrate modes (2.0 - 4.0 kbit/s) in a scalable and a non-scalable scheme, a variable bitrate mode (< 2.0 kbit/s) and the functionality of pitch and speed change. Only 8 kHz sampling rate and mono audio channel are supported.

1.5.1.2.11 TTSI object type

The TTSI object is supported by the TTSI tools described in subpart 6. It allows very-low-bitrate phonemic descriptions of speech to be transmitted in the bitstream payload and then synthesized into sound. No particular speech synthesis method is specified in MPEG-4; rather, the TTSI tools specify an *interface* to non-normative synthesis methods. This method has a bit rate ranging 200 ~ 1200 bit/s. The TTSI object also supports synchronization of the synthesized speech with the facial animation defined in ISO/IEC 14496-2.

1.5.1.2.12 Main Synthetic object type

The main synthetic object allows the use of all MPEG-4 Structured Audio tools (described in subpart 5 of this standard). It supports flexible, high-quality algorithmic synthesis using the SAOL music-synthesis language; efficient wavetable synthesis with the SASBF sample-bank format; and enables the use of high-quality mixing and postproduction in the Systems AudioBIFS toolset. Sound can be described at 0 kbit/s (no continuous cost) to 3-4 kbit/s for extremely expressive sounds in the MPEG-4 Structured Audio format.

1.5.1.2.13 Wavetable Synthesis object type

The wavetable synthesis object is supported only by the SASBF format and MIDI tools. It allows the use of simple „sampling synthesis“ in presentations where the quality and flexibility of the full synthesis toolset is not required.

1.5.1.2.14 General Midi object type

The General MIDI object is included only to provide interoperability with existing content. Normative sound quality and decoder behavior are not provided with the General MIDI object.

1.5.1.2.15 Algorithmic Synthesis and Audio FX object type

The Algorithmic Synthesis object provides SAOL-based synthesis capabilities for very low-bitrate terminals. It is also used to support the AudioBIFS **AudioFX** node in content where sound synthesis capability is not needed.

1.5.1.2.16 Error Resilient (ER) AAC Low Complexity (LC) object type

The Error Resilient (ER) MPEG-4 AAC Low Complexity object type is the counterpart to the MPEG-4 AAC Low Complexity object, with additional error resilient functionality.

1.5.1.2.17 Error Resilient (ER) AAC Long Term Predictor (LTP) object type

The Error Resilient (ER) MPEG-4 AAC LTP object type is the counterpart to the MPEG-4 AAC LTP object, with additional error resilient functionality.

1.5.1.2.18 Error Resilient (ER) AAC scalable object type

The Error Resilient (ER) MPEG-4 AAC scalable object type is the counterpart to the MPEG-4 AAC scalable object, with additional error resilient functionality.

1.5.1.2.19 Error Resilient (ER) TwinVQ object type

The Error Resilient (ER) TwinVQ object type is the counterpart to the MPEG-4 TwinVQ object, with additional error resilient functionality.

1.5.1.2.20 Error Resilient (ER) BSAC object type

The ER BSAC object is supported by the fine grain scalablility tool (BSAC: Bit-Sliced Arithmetic Coding). It provides error resilience as well as fine step scalability in the MPEG-4 General Audio (GA) coder. It is used in combination with the AAC coding tools and replaces the noiseless coding and the bitstream payload formatting of MPEG-4 AAC coder. A large number of scalable layers are available, providing 1 kbit/s/ch enhancement layer, i.e. 2 kbit/s steps for a stereo signal.

1.5.1.2.21 Error Resilient (ER) AAC LD object type

The AAC LD object is supported by the low delay AAC coding tool. It also permits combinations with the PNS tool and the LTP tool. AAC LD object provides the ability to extend the usage of generic low bitrate audio coding to applications requiring a very low delay of the encoding / decoding chain (e.g. full-duplex real-time communications).

1.5.1.2.22 Error Resilient (ER) CELP object type

The ER CELP object is supported by silence compression and ER tools. It provides the ability to reduce the average bitrate thanks to a lower-bitrate compression for silence, with additional error resilient functionality.

1.5.1.2.23 Error Resilient (ER) HVXC object type

The ER HVXC object is supported by the parametric speech coding (HVXC) tools, which provide fixed bitrate modes (2.0-4.0 kbit/s) and variable bitrate modes (< 2.0 kbit/s and < 4.0 kbit/s) both in a scalable and a non-scalable scheme, and the functionality of pitch and speed change. The syntax to be used with the EP-Tool, and the error concealment functionality are supported for the use for error-prone channels. Only 8 kHz sampling rate and mono audio channel are supported.

1.5.1.2.24 Error Resilient (ER) HILN object type

The ER HILN object is supported by the parametric audio coding tools (HILN: Harmonic and Individual Lines plus Noise) which provide coding of general audio signals at very low bitrates ranging from below 4 kbit/s to above 16 kbit/s. Bitrate scalability and the functionality of speed and pitch change are available. The ER HILN object supports mono audio objects at a wide range of sampling rates.

1.5.1.2.25 Error Resilient (ER) Parametric object type

The ER Parametric object is supported by the parametric audio coding and speech coding tools HILN and HVXC. This integrated parametric coder combines the functionalities of the ER HILN and the ER HVXC objects. Only 8 kHz sampling rate and mono audio channel are supported.

1.5.1.2.26 SSC Audio object type

The SSC (SinuSoidal Coding) object combines the SSC parametric coding tools: transients, sinusoids, noise and parametric stereo. Only 44.1kHz and coding of mono, dual mono and (parametric) stereo are supported.

1.5.1.2.27 Layer-1 Audio object type

The Layer-1 object is the counterpart of the audio coding scheme Layer-1 specified in ISO/IEC 11172-3 and 13818-3.

1.5.1.2.28 Layer-2 Audio object type

The Layer-2 object is the counterpart of the audio coding scheme Layer-2 specified in ISO/IEC 11172-3 and ISO/IEC 13818-3.

1.5.1.2.29 Layer-3 Audio object type

The Layer-3 object is very similar to the audio coding scheme Layer-3 specified in ISO/IEC 11172-3 and ISO/IEC 13818-3. However, the use of Layer 3 encoded data as defined in ISO/IEC 13818-3 is limited to the “Lower

Sampling Frequencies" case, i.e. the Layer 3 multi-channel syntax defined in ISO/IEC 13818-3 is not permitted in this scope. Furthermore, additional sampling rates have been specified.

1.5.2 Audio profiles and levels

1.5.2.1 Profiles

The following Audio Profiles have been defined:

1. The **Speech Audio Profile** provides a parametric speech coder, a CELP speech coder and a Text-To-Speech interface.
2. The **Synthetic Audio Profile** provides the capability to generate sound and speech at very low bitrates.
3. The **Scalable Audio Profile,** a superset of the Speech Audio Profile, is suitable for scalable coding of speech and music, for transmission methods such as Internet and Digital Broadcasting.
4. The **Main Audio Profile** is a superset of the scalable profile, the speech profile and the synthesis profile, containing tools for natural and synthetic audio.
5. The **High Quality Audio Profile** contains the CELP speech coder and the Low Complexity AAC coder including Long Term Prediction. Scalable coding coding can be performed by the AAC Scalable object type. Optionally, the new error resilient (ER) bitstream payload syntax may be used.
6. The **Low Delay Audio Profile** contains the HVXC and CELP speech coders (optionally using the ER bitstream payload syntax), the low-delay AAC coder and the Text-to-Speech interface TTSI.
7. The **Natural Audio Profile** contains all natural audio coding tools available in MPEG-4.
8. The **Mobile Audio Internetworking Profile** contains the low-delay and scalable AAC object types including TwinVQ and BSAC. This profile is intended to extend communication applications using non-MPEG speech coding algorithms with high quality audio coding capabilities.
9. The **AAC Profile** contains the audio object type 2 (AAC-LC).
10. The **High Efficiency AAC Profile** contains the audio object types 5 (SBR) and 2 (AAC LC). The High Efficiency AAC Profile is a superset of the AAC Profile.

Table 1.3 — Audio Profiles definition

Object Type ID	Audio Object Type	Speech Audio Profile	Synthetic Audio Profile	Scalable Audio Profile	Main Audio Profile	High Quality Audio Profile	Low Delay Audio Profile	Natural Audio Profile	Mobile Audio Internetworking Profile	AAC Profile	High Efficiency AAC Profile
0	Null										
1	AAC main				X			X			
2	AAC LC			X	X	X		X		X	X
3	AAC SSR				X			X			
4	AAC LTP			X	X	X		X			
5	SBR										X
6	AAC Scalable			X	X	X		X			
7	TwinVQ			X	X			X			
8	CELP	X		X	X	X	X	X			
9	HVXC	X		X	X		X	X			
10	(reserved)										
11	(reserved)										
12	TTSI	X	X	X	X		X	X			
13	Main synthetic		X		X						
14	Wavetable synthesis		(Subset of Main Synthetic)		(Subset of Main Synthetic)						

15	General MIDI		(Subset of Main Synthetic)		(Subset of Main Synthetic)						
16	Algorithmic Synthesis and Audio FX		(Subset of Main Synthetic)		(Subset of Main Synthetic)						
17	ER AAC LC					X		X	X		
18	(reserved)										
19	ER AAC LTP					X		X			
20	ER AAC Scalable					X		X	X		
21	ER TwinVQ							X	X		
22	ER BSAC							X	X		
23	ER AAC LD						X	X	X		
24	ER CELP					X	X	X			
25	ER HVXC						X	X			
26	ER HILN							X			
27	ER Parametric							X			
28	SSC										
29	(reserved)										
30	(reserved)										
31	(escape)										
32	Layer-1										
33	Layer-2										
34	Layer-3										
35	DST										

In addition to the profile descriptions given above it is stated that AAC Scalable objects using wide-band CELP core layer (with or without ER bitstream payload syntax) are **not** part of any Audio Profile.

1.5.2.2 Complexity units

Complexity units are defined to give an approximation of the decoder complexity in terms of processing power and RAM usage required for processing MPEG-4 Audio bitstream payloads in dependence of specific parameters.

The approximated processing power is given in “Processor Complexity Units” (PCU), specified in integer numbers of MOPS. The approximated RAM usage is given in “RAM Complexity Units” (RCU), specified in mostly integer numbers of kWords (1000 words). The RCU numbers do not include working buffers that can be shared between different objects and/or channels.

If a profile level is specified by the maximum number of complexity units, then a flexible configuration of the decoder handling different types of objects is allowed under the constraint that both values for the total complexity for decoding and sampling rate conversion (if needed) do not exceed this limit.

The following table gives complexity estimates for the different object types. PCU values are given in MOPS per channel, RCU values in kWords per channel (with respect to AAC, channel refers to Main channel, e. g. the channel of a SCE, one channel of a CPE, or the channel of an independently switched CCE).

Table 1.4 — Complexity of Audio Object Types and SR conversion

Object Type	Parameters	PCU (MOPS)	RCU	Remark
AAC Main	fs = 48 kHz	5	5	1)
AAC LC	fs = 48 kHz	3	3	1)
AAC SSR	fs = 48 kHz	4	3	1)
AAC LTP	fs = 48 kHz	4	4	1)
SBR	fs = 24/48 kHz (in/out) (SBR tool)	3	2.5	1)
	fs = 24/48 kHz (in/out) (Low Power SBR tool)	2	1.5	1)
	fs = 48/48 kHz (in/out) (Down Sampled SBR tool)	4.5	2.5	1)
	fs = 48/48 kHz (in/out) (Low Power Down Sampled SBR tool)	3	1.5	1)
AAC Scalable	fs = 48 kHz	5	4	1), 2)
TwinVQ	fs = 24 kHz	2	3	1)
CELP	fs = 8 kHz	1	1	
CELP	fs = 16 kHz	2	1	
CELP	fs = 8/16 kHz (bandwidth scalable)	3	1	
HVXC	fs = 8 kHz	2	1	
TTSI		-	-	4)
General MIDI		4	1	
Wavetable Synthesis	fs = 22.05 kHz	depends on bitstream payloads (3)	depends on bitstream payloads (3)	
Main Synthetic		depends on bitstream	depends on bitstream	
Algorithmic Synthesis and AudioFX		depends on bitstream	depends on bitstream	
Sampling Rate Conversion	rf = 2, 3, 4, 6, 8, 12	2	0.5	7)
ER AAC LC	fs = 48 kHz	3	3	1)
ER AAC LTP	fs = 48 kHz	4	4	1)
ER AAC Scalable	fs = 48 kHz	5	4	1), 2)
ER TwinVQ	fs = 24 kHz	2	3	1)
ER BSAC	fs = 48 kHz (input buffer size=26000bits)	4	4	1)
	fs = 48 kHz (input buffer size=106000bits)	4	8	
ER AAC LD	fs = 48 kHz	3	2	1)
ER CELP	fs = 8 kHz	2	1	
	fs = 16 kHz	3	1	
ER HVXC	fs = 8 kHz	2	1	
ER HILN	fs = 16 kHz, ns=93	15	2	6)
	fs = 16 kHz, ns=47	8	2	
ER Parametric	fs = 8 kHz, ns=47	4	2	5),6)

Definitions:

- fs = sampling frequency
- rf = ratio of sampling rates

Notes:

1) PCU proportional to sampling frequency.
2) Includes core decoder.

3) See ISO/IEC 14496-4.
4) The complexity for speech synthesis is not taken into account.
5) Parametric coder in HILN mode, for HVXC mode see ER HVXC.
6) PCU depends on fs and ns, see below.
7) Sampling Rate Conversion is needed, if objects of different sampling rates are combined in a scene (see ISO/IEC 14496-1:2000, subclause 9.2.2.13.2.1). The specified values have to be added for each required conversion.

PCU for HILN:

The computational complexity of HILN depends on the sampling frequency fs and the maximum number of sinusoids ns to be synthesized simultaneously. The value of ns for a frame is the total number of harmonic and individual lines synthesized in that frame, i.e. the number of starting plus continued plus ending lines. For fs in kHz, the PCU in MOPS is calculated as follows:

PCU = (1 + 0.15*ns) * fs/16

The typical maximum values of ns are 47 for 6 kbit/s HILN and 93 for 16 kbit/s HILN streams.

PCU and RCU for AAC:

For AAC object types, PCU and RCU depend on sampling frequency and channel configuration in the following way:

PCUs

PCU = (fs / fs_ref) * PCU_ref * (2 * #CPE + #SCE + #LFE + #IndepCouplingCh + 0.3 * #DepCouplingCh)

fs: actual sampling frequency

fs_ref: reference sampling frequency (sampling freqency for the given PCU_ref)

PCU_ref: reference PCU given in Table 1.4

#SCE: number of SCEs

#CPE: ...

RCUs

#CPE < 2:

RCU = RCU_ref * [#SCE + 0.5 * #LFE + 0.5 * #IndepCouplingCh + 0.4 * #DepCouplingCh]
+ [RCU_ref + (RCU_ref – 1)] * #CPE

#CPE >= 2:

RCU = RCU_ref * [#SCE + 0.5 * #LFE + 0.5 * #IndepCouplingCh + 0.4 * #DepCouplingCh]
+ [RCU_ref + (RCU_ref – 1) * (2 * #CPE – 1)]

RCU_ref: reference RCU given in Table 1.4

#SCE: number of SCEs

#CPE: ...

1.5.2.3 Levels within the profiles

The notation used to specify the number of audio channels indicates the number of main audio channels. Based on the number of main audio channels (A), Table 1.5 indicates for object types derived from the AAC multichannel syntax the number of LFE channels (L), the number of independently switched coupling channels (I), and the number of dependently switched coupling channels (D) in the form A.L.I.D.

Table 1.5 — Maximum number of the individual AAC channel types depending on the specified number of main audio channels

	Number of Main Audio Channels		
AOT	1	2	5
1 (AAC main)	1.0.0.0	2.0.0.0	5.1.1.1
2 (AAC LC)	1.0.0.0	2.0.0.0	5.1.0.1
3 (AAC SSR)	1.0.0.0	2.0.0.0	5.1.0.0
4 (AAC LTP)	1.0.0.0	2.0.0.0	5.1.0.1
17 (ER AAC LC)	1.0.0.0	2.0.0.0	5.1.0.0
19 (ER AAC LTP)	1.0.0.0	2.0.0.0	5.1.0.0
23 (ER AAC LD)	1.0.0.0	2.0.0.0	5.1.0.0

Note: In the case of scalable coding schemes, only the first instantiation of each object type will be counted to determine the number of objects relevant to the level definition and complexity metric. For example, in a scalable coder consisting of a CELP core coder and two enhancement layers implemented by means of AAC scalable objects, one CELP object and one AAC scalable object and their associated complexity metrics are counted since there is almost no overhead associated with the second (and any further) GA enhancement layer.

- **Levels for Speech Audio Profile**

Two levels are defined by number of objects:

1. One speech object.
2. Up to 20 speech objects.

- **Levels for Synthetic Audio Profile**

Three levels are defined :

1. Synthetic Audio 1: All bitstream payload elements may be used with:
   - “Low processing” (exact numbers in ISO/IEC 14496-4:2000)
   - Only core sample rates may be used
   - No more than one TTSI object
2. Synthetic Audio 2: All bitstream payload elements may be used with:
   - ”Medium processing” (exact numbers in ISO/IEC 14496-4:2000).
   - Only core sample rates may be used.
   - no more than four TTSI objects.
3. Synthetic Audio 3: All bitstream payload elements may be used with:
   - ”High processing” (exact numbers in ISO/IEC 14496-4:2000).
   - no more than twelve TTSI objects.

- **Levels for Scalable Audio Profile**

Four levels are defined by configuration; complexity units define the fourth level:

1. Maximum 24 kHz of sampling rate, one mono object (all object Types).
2. Maximum 24 kHz of sampling rate, one stereo object or two mono objects (all object Types).
3. Maximum 48 kHz of sampling rate, one stereo object or two mono objects (all object Types).
4. Maximum 48 kHz of sampling rate, one object with 5 main channels or multiple objects with at maximum one integer factor sampling rate conversion for a maximum of two channels.
   Flexible configuration is allowed with PCU < 30 and RCU < 19.

For the audio object types 2 (AAC LC), and 4 (AAC LTP), only a frame length of 1024 samples is permitted with respect to level 1, 2, 3 and 4. For the audio object types 2 (AAC LC) and 4 (AAC LTP), mono or stereo mixdown elements are not permitted with respect to level 1, 2, 3 and 4. For the audio object type 6 (AAC Scalable) the following restrictions apply. The number of AAC layers shall not exceed 8 in any scalable configuration. If audio object type 8 (CELP) is used as core layer coder, the number of CELP layers shall not exceed 2. If audio object type 7 (TwinVQ) is used as base layer coder, only one mono TwinVQ layer is permitted.

- **Levels for Main Audio Profile**

Main Audio Profile contains all natural and synthetic object types. Levels are then defined as a combination of the two different types of levels from the two different metrics defined for natural tools (computation-based metrics) and synthetic tools (macro-oriented metrics).

For Object Types not belonging to the Synthetic Profile four levels are defined:

1. Natural Audio 1: PCU < 40, RCU < 20
2. Natural Audio 2: PCU < 80, RCU < 64
3. Natural Audio 3: PCU < 160, RCU < 128
4. Natural Audio 4: PCU < 320, RCU < 256

For Object Types belonging to the Synthetic Profile the same three Levels are defined as above, i.e. Synthetic Audio 1, Synthetic Audio 2 and Synthetic Audio 3.

Four Levels are then defined for Main Profile:

1. Natural Audio 1 + Synthetic Audio 1
2. Natural Audio 2 + Synthetic Audio 1
3. Natural Audio 3 + Synthetic Audio 2
4. Natural Audio 4 + Synthetic Audio 3

For the audio object types 1 (AAC main), 2 (AAC LC), 3 (AAC SSR), and 4 (AAC LTP), only a frame length of 1024 samples is permitted with respect to level 1, 2 , 3 and 4. For the audio object types 1 (AAC Main), 2 (AAC LC), 3 (AAC SSR) and 4 (AAC LTP), mono or stereo mixdown elements are not permitted with respect to level 1, 2 , 3 and 4. For the audio object type 6 (AAC Scalable) the following restrictions apply. The number of AAC layers shall not exceed 8 in any scalable configuration. If audio object type 8 (CELP) is used as core layer coder, the number of CELP layers shall not exceed 2. If audio object type 7 (TwinVQ) is used as base layer coder, only one mono TwinVQ layer is permitted.

- **Levels for the High Quality Audio Profile**

Table 1.6 — Levels for the High Quality Audio Profile

Level	Max. channels/object	Max. sampling rate [kHz]	Max PCU [*2]	Max RCU [*2]	EP-Tool: Max. redundancy by class FEC [*1]	EP-Tool: Max. # stages of interleaving per object
1	2	22.05	5	8	0 %	0
2	2	48	10	8	0 %	0
3	5	48	25	12 [*3]	0 %	0
4	5	48	100	42 [*3]	0 %	0
5	2	22.05	5	8	20%	9
6	2	48	10	8	20%	9
7	5	48	25	12 [*3]	20%	22
8	5	48	100	42 [*3]	20%	22

*1: The value specifies the maximum redundancy based on the available audio object with the longest maximum frame length. The redundancy might be larger in the case of smaller frame lengths. However, the application of any class FEC is not permitted if 0 % is specified. The limit is valid independently for each audio object. Since this value does neither cover the EP header and its protection nor any CRC, another 5 % must always be added to this value in order to derive the necessary increase of the minimum decoder input buffer. This does imply, that not more than those 5 % might be spent for the EP header and its protection or any CRC.
*2: Level 5 to 8 do not include RAM and computational complexity for the EP tool.
*3: Sharing of work buffers between multiple objects or channel pair elements is assumed.

For the audio object types 2 (AAC LC), 4 (AAC LTP), 17 (ER AAC LC), and 19 (ER AAC LTP) only a frame length of 1024 samples is permitted with respect to level 1, 2, 3, 4, 5, 6, 7 and 8. For the audio object types 2 (AAC LC) and 4 (AAC LTP), mono or stereo mixdown elements are not permitted with respect to level 1, 2, 3, 4, 5, 6, 7 and 8. For the audio object type 6 and 20 ((ER) AAC Scalable) the following restrictions apply. The number of AAC layers shall not exceed 8 in any scalable configuration. If audio object type 8 or 24 ((ER) CELP) is used as core layer coder, the number of CELP layers shall not exceed 2.

- **Levels for the Low Delay Audio Profile**

Table 1.7 — Levels for the Low Delay Audio Profile

Level	Max. channels/object	Max. sampling rate [kHz]	Max PCU *2	Max RCU *2	EP-Tool: Max. redundancy by class FEC *1	EP-Tool: Max. # stages of interleaving per object
1	1	8	2	1	0 %	0
2	1	16	3	1	0 %	0
3	1	48	3	2	0 %	0
4	2	48	24	12 *3	0 %	0
5	1	8	2	1	100%	5
6	1	16	3	1	100%	5
7	1	48	3	2	20%	5
8	2	48	24	12 *3	20%	9

*1: The value specifies the maximum redundancy based on the available audio object with the longest maximum frame length. The redundancy might be larger in the case of smaller frame lengths. However, the application of any class FEC is not permitted if 0 % is specified. The limit is valid independently for each audio object. Since this value does neither cover the EP header and its protection nor any CRC, another 5 % must always be added to this value in order to derive the necessary increase of the minimum decoder input buffer. This does imply, that not more than those 5 % might be spent for the EP header and its protection or any CRC.
*2: Level 5 to 8 do not include RAM and computational complexity for the EP tool.
*3: Sharing of work buffers between multiple objects or channel pair elements is assumed.

- **Levels for the Natural Audio Profile**

Table 1.8 — Levels for the Natural Audio Profile

Level	Max. sampling rate [kHz]	Max PCU *2	EP-Tool: Max. redundancy by class FEC *1	EP-Tool: Max. # stages of interleaving per object
1	48	20	0 %	0
2	96	100	0 %	0
3	48	20	20%	9
4	96	100	20%	22

*1: The value specifies the maximum redundancy based on the available audio object with the longest maximum frame length. The redundancy might be larger in the case of smaller frame lengths. However, the application of any class FEC is not permitted if 0 % is specified. The limit is valid independently for each audio object.

Since this value does neither cover the EP header and its protection nor any CRC, another 5 % must always be added to this value in order to derive the necessary increase of the minimum decoder input buffer. This does imply, that not more than those 5 % might be spent for the EP header and its protection or any CRC.

*2: Level 3 and 4 do not include computational complexity for the EP tool.

No RCU limitations are specified for this profile.

For the audio object types 1 (AAC main), 2 (AAC LC), 3 (AAC SSR), 4 (AAC LTP), 17 (ER AAC LC), and 19 (ER AAC LTP) only a frame length of 1024 samples is permitted with respect to level 1, 2 , 3 and 4. For the audio object types 1 (AAC Main), 2 (AAC LC), 3 (AAC SSR) and 4 (AAC LTP), mono or stereo mixdown elements are not permitted with respect to level 1, 2 , 3 and 4. For the audio object type 6 and 20 ((ER)AAC Scalable) the following restrictions apply. The number of AAC layers shall not exceed 8 in any scalable configuration. If audio object type 8 or 24 ((ER) CELP) is used as core layer coder, the number of CELP layers shall not exceed 2. If audio object type 7 or 21 ((ER) TwinVQ) is used as base layer coder, only one mono TwinVQ layer is permitted.

- **Levels for the Mobile Audio Internetworking Profile**

Table 1.9 — Levels for the Mobile Audio Internetworking Profile

Level	Max. channels/object	Max. sampling rate [kHz]	Max PCU *3	Max RCU *2 *3	Max. # audio objects	EP-Tool: Max. redundancy by class FEC *1	EP-Tool: Max. # stages of interleaving per object
1	1	24	2.5	4	1	0 %	0
2	2	48	10	8	2	0 %	0
3	5	48	25	12 *4	-	0 %	0
4	1	24	2.5	4	1	20%	5
5	2	48	10	8	2	20%	9
6	5	48	25	12 *4	-	20%	22

*1: The value specifies the maximum redundancy based on the available audio object with the longest maximum frame length. The redundancy might be larger in the case of smaller frame lengths. However, the application of any class FEC is not permitted if 0 % is specified. The limit is valid independently for each audio object. Since this value does neither cover the EP header and its protection nor any CRC, another 5 % must always be added to this value in order to derive the necessary increase of the minimum decoder input buffer. This does imply, that not more than those 5 % might be spent for the EP header and its protection or any CRC.

*2: The maximum RCU for one channnel in any object in this profile is 4. For the ER BSAC, this limits the input buffer size. The maximum possible input buffer size in bits for this case is given in PCU/RCU (Table 1.4).

*3: Level 4 to 6 do not include RAM and computational complexity for the EP tool.

*4: Sharing of work buffers between multiple objects or channel pair elements are assumed.

For the audio object type 17 (ER AAC LC) only a frame length of 1024 samples is permitted with respect to level 1,2,3,4,5 and 6. For the audio object type 20 (ER AAC Scalable) the following restrictions apply. The number of AAC layers shall not exceed 8 in any scalable configuration. If audio object type 21 (ER TwinVQ) is used as base layer coder, only one mono TwinVQ layer is permitted.

- **Levels for the AAC Profile**

Table 1.10 — Levels for the AAC Profile

Level	Max. channels/ object	Max. sampling rate [kHz]	Max. PCU	Max. RCU
1	2	24	3	5
2	2	48	6	5
3	NA	NA	NA	NA
4	5	48	19	15
5	5	96	38	15

For the audio object type 2 (AAC LC), mono or stereo mixdown elements are not permitted.

The NA (Not Applicable) levels are introduced to emphasize the hierarchical structure of the AAC Profile and the High Efficiency AAC Profile. Hence, a decoder supporting the High Efficiency AAC Profile at a given level can decode an AAC Profile stream of the same or a lower level. The NA levels are not indicated in the audioProfileLevelIndication table (Table 1.12).

- **Levels for the High Efficiency AAC Profile**

Table 1.11 — Levels for the High Efficiency AAC Profile

Level	Max. channels/ object	Max. AAC sampling rate, SBR not present [kHz]	Max. AAC sampling rate, SBR present [kHz]	Max. SBR sampling rate [kHz] (in/out)	Max. PCU	Max. RCU	Max. PCU Low power SBR	Max. RCU Low power SBR
1	NA	NA	NA	NA	NA	NA	NA	NA
2	2	48	24	24/48	9	10	7	8
3	2	48	48	48/48 (Note 1)	15	10	12	8
4	5	48	24/48 (Note 2)	48/48 (Note 1)	25	28	20	23
5	5	96	48	48/96	49	28	39	23
Note 1: For level 3 and level 4 decoders, it is mandatory to operate the SBR tool in downsampled mode if the sampling rate of the AAC core is higher than 24kHz. Hence, if the SBR tool operates on a 48kHz AAC signal, the internal sampling rate of the SBR tool will be 96kHz, however, the output signal will be downsampled by the SBR tool to 48kHz. Note 2: For one or two channels the maximum AAC sampling rate, with SBR present, is 48kHz. For more than two channels the maximum AAC sampling rate, with SBR present, is 24kHz.								

For the audio object type 2 (AAC LC), mono or stereo mixdown elements are not permitted.

1.5.2.4 audioProfileLevelIndication

audioProfileLevelIndication is a data element of the InitialObjectDescriptor defined in ISO/IEC 14496-1. It indicates the profile and level required to process the content associated with this InitialObjectDesriptor as defined in Table 1.12.

Table 1.12 — audioProfileLevelIndication values

Value	Profile	Level
0x00	Reserved for ISO use	-
0x01	Main Audio Profile	L1
0x02	Main Audio Profile	L2
0x03	Main Audio Profile	L3
0x04	Main Audio Profile	L4
0x05	Scalable Audio Profile	L1
0x06	Scalable Audio Profile	L2
0x07	Scalable Audio Profile	L3
0x08	Scalable Audio Profile	L4
0x09	Speech Audio Profile	L1
0x0A	Speech Audio Profile	L2
0x0B	Synthetic Audio Profile	L1
0x0C	Synthetic Audio Profile	L2
0x0D	Synthetic Audio Profile	L3
0x0E	High Quality Audio Profile	L1
0x0F	High Quality Audio Profile	L2
0x10	High Quality Audio Profile	L3
0x11	High Quality Audio Profile	L4
0x12	High Quality Audio Profile	L5
0x13	High Quality Audio Profile	L6
0x14	High Quality Audio Profile	L7
0x15	High Quality Audio Profile	L8
0x16	Low Delay Audio Profile	L1
0x17	Low Delay Audio Profile	L2
0x18	Low Delay Audio Profile	L3
0x19	Low Delay Audio Profile	L4
0x1A	Low Delay Audio Profile	L5
0x1B	Low Delay Audio Profile	L6
0x1C	Low Delay Audio Profile	L7
0x1D	Low Delay Audio Profile	L8
0x1E	Natural Audio Profile	L1
0x1F	Natural Audio Profile	L2
0x20	Natural Audio Profile	L3
0x21	Natural Audio Profile	L4
0x22	Mobile Audio Internetworking Profile	L1
0x23	Mobile Audio Internetworking Profile	L2
0x24	Mobile Audio Internetworking Profile	L3
0x25	Mobile Audio Internetworking Profile	L4
0x26	Mobile Audio Internetworking Profile	L5
0x27	Mobile Audio Internetworking Profile	L6
0x28	AAC Profile	L1
0x29	AAC Profile	L2
0x2A	AAC Profile	L4
0x2B	AAC Profile	L5
0x2C	High Efficiency AAC Profile	L2
0x2D	High Efficiency AAC Profile	L3
0x2E	High Efficiency AAC Profile	L4
0x2F	High Efficiency AAC Profile	L5
0x30 - 0x7F	reserved for ISO use	-
0x80 - 0xFD	user private	-
0xFE	no audio profile specified	-
0xFF	no audio capability required	-
NOTE — Usage of the value 0xFE indicates that the content described by this InitialObjectDescriptor does not comply to any audio profile specified in ISO/IEC 14496-3.		

Usage of the value 0xFF indicates that none of the audio profile capabilities are required for this content.

1.6 Interface to ISO/IEC 14496-1 (MPEG-4 Systems)

1.6.1 Introduction

The header streams are transported via MPEG-4 systems. These streams contain configuration information, which is necessary for the decoding process and parsing of the raw data streams. However, an update is only necessary if there are changes in the configuration.

The payloads contain all information varying on a frame to frame basis and therefore carry the actual audio information.

1.6.2 Syntax

1.6.2.1 AudioSpecificConfig

AudioSpecificConfig() extends the abstract class DecoderSpecificInfo, as defined in ISO/IEC 14496-1, when DecoderConfigDescriptor.objectTypeIndication refers to streams complying with ISO/IEC 14496-3 in this case the existence of AudioSpecificConfig() is mandatory.

Table 1.13 — Syntax of AudioSpecificConfig()

Syntax	No. of bits	Mnemonic
AudioSpecificConfig ()		
{		
audioObjectType = GetAudioObjectType();		
samplingFrequencyIndex;	**4**	**bslbf**
if (samplingFrequencyIndex == 0xf) {		
samplingFrequency;	**24**	**uimsbf**
}		
channelConfiguration;	**4**	**bslbf**
sbrPresentFlag = -1;		
if (audioObjectType == 5) {		
extensionAudioObjectType = audioObjectType;		
sbrPresentFlag = 1;		
extensionSamplingFrequencyIndex;	**4**	**uimsbf**
if (extensionSamplingFrequencyIndex == 0xf)		
extensionSamplingFrequency;	**24**	**uimsbf**
audioObjectType = GetAudioObjectType();		
}		
else {		
extensionAudioObjectType = 0;		
}		
switch (audioObjectType) {		

```
    case 1:
    case 2:
    case 3:
    case 4:
    case 6:
    case 7:
    case 17:
    case 19:
    case 20:
    case 21:
    case 22:
    case 23:
        GASpecificConfig();
        break:
    case 8:
        CelpSpecificConfig();
        break;
    case 9:
        HvxcSpecificConfig();
        break:
    case 12:
        TTSSpecificConfig();
        break;
    case 13:
    case 14:
    case 15:
    case 16:
        StructuredAudioSpecificConfig();
        break;
    case 24:
        ErrorResilientCelpSpecificConfig();
        break;
    case 25:
        ErrorResilientHvxcSpecificConfig();
        break;
    case 26:
    case 27:
        ParametricSpecificConfig();
        break;
    case 28:
        SSCSpecificConfig();
        break;
    case 32:
    case 33:
    case 34:
        MPEG_1_2_SpecificConfig();
        break;
    case 35:
        DSTSpecificConfig();
        break;
    default:
        /* reserved */
    }
    switch (audioObjectType) {
```

```
    case 17:
    case 19:
    case 20:
    case 21:
    case 22:
    case 23:
    case 24:
    case 25:
    case 26:
    case 27:
        epConfig;                                                    2        bslbf
        if ( epConfig == 2 || epConfig == 3 ) {
            ErrorProtectionSpecificConfig();
        }
        if ( epConfig == 3 ) {
            directMapping;                                           1        bslbf
            if ( ! directMapping ) {
                /* tbd */
            }
        }
    }
    if ( extensionAudioObjectType != 5 && bits_to_decode() >= 16 ) {
        syncExtensionType;                                           11       bslbf
        if (syncExtensionType == 0x2b7) {
            extensionAudioObjectType = GetAudioObjectType();
            if ( extensionAudioObjectType == 5 ) {
                sbrPresentFlag;                                      1        uimsbf
                if (sbrPresentFlag == 1) {
                    extensionSamplingFrequencyIndex;                 4        uimsbf
                    if ( extensionSamplingFrequencyIndex == 0xf )
                        extensionSamplingFrequency;                  24       uimsbf
                }
            }
        }
    }
}
```

Table 1.14 — Syntax of GetAudioObjectType()

Syntax	No. of bits	Mnemonic
GetAudioObjectType()		
{		
audioObjectType;	**5**	**uimsbf**
if (audioObjectType == 31) {		
audioObjectType = 32 + **audioObjectTypeExt**;	**6**	**uimsbf**
}		
return audioObjectType;		
}		

The classes defined in subclauses 1.6.2.1.1 to 1.6.2.1.9 do not extend the BaseDescriptor class (see ISO/IEC 14496-1) and consequently their length shall be derived by difference from the length of AudioSpecificConfig().

1.6.2.1.1 HvxcSpecificConfig

Defined in ISO/IEC 14496-3 subpart 2.

1.6.2.1.2 CelpSpecificConfig

Defined in ISO/IEC 14496-3 subpart 3.

1.6.2.1.3 GASpecificConfig

Defined in ISO/IEC 14496-3 subpart 4.

1.6.2.1.4 StructuredAudioSpecificConfig

Defined in ISO/IEC 14496-3 subpart 5.

1.6.2.1.5 TTSSpecificConfig

Defined in ISO/IEC 14496-3 subpart 6.

1.6.2.1.6 ParametricSpecificConfig

Defined in ISO/IEC 14496-3 subpart 7.

1.6.2.1.7 ErrorProtectionSpecificConfig

Defined in subclause 1.8.2.1.

1.6.2.1.8 ErrorResilientCelpSpecificConfig

Defined in ISO/IEC 14496-3 subpart 3.

1.6.2.1.9 ErrorResilientHvxcSpecificConfig

Defined in ISO/IEC 14496-3 subpart 2.

1.6.2.1.10 MPEG_1_2_SpecificConfig

Defined in ISO/IEC 14496-3 subpart 9.

1.6.2.1.11 SSCSpecificConfig

Defined in ISO/IEC 14496-3 subpart 8.

1.6.2.2 Payloads

1.6.2.2.1 Overview

For the NULL object the payload shall be 16 bit signed integer in the range from -32768 to +32767. The payloads for all other audio object types are defined in the corresponding parts. These are the basic entities to be carried by the systems transport layer. Note that for all natural audio coding schemes the output is scaled for a maximum of 32767/-32768. However, the MPEG-4 System compositor expects a scaling.

Payloads that are not byte aligned are padded at the end for transport schemes which require byte alignment.

The following table shows an overview about where the Elementary Stream payloads for the Audio Object Types can be found and where the detailed syntax is defined.

Table 1.15 — Audio Object Types

Object Type ID	Audio Object Type	definition of elementary stream payloads and detailed syntax	Mapping of audio payloads to access units and elementary streams
0	NULL		
1	AAC MAIN	ISO/IEC 14496-3 subpart 4	see subclause 1.6.2.2.2.1.2
2	AAC LC	ISO/IEC 14496-3 subpart 4	see subclause 1.6.2.2.2.1.2
3	AAC SSR	ISO/IEC 14496-3 subpart 4	see subclause 1.6.2.2.2.1.2
4	AAC LTP	ISO/IEC 14496-3 subpart 4	see subclause 1.6.2.2.2.1.2
5	SBR	ISO/IEC 14496-3 subpart 4	
6	AAC scalable	ISO/IEC 14496-3 subpart 4	see subclause 1.6.2.2.2.1.3
7	TwinVQ	ISO/IEC 14496-3 subpart 4	
8	CELP	ISO/IEC 14496-3 subpart 3	
9	HVXC	ISO/IEC 14496-3 subpart 2	
10	(reserved)		
11	(reserved)		
12	TTSI	ISO/IEC 14496-3 subpart 6	
13	Main synthetic	ISO/IEC 14496-3 subpart 5	
14	Wavetable synthesis	ISO/IEC 14496-3 subpart 5	
15	General MIDI	ISO/IEC 14496-3 subpart 5	
16	Algorithmic Synthesis and Audio FX	ISO/IEC 14496-3 subpart 5	
17	ER AAC LC	ISO/IEC 14496-3 subpart 4	see subclause 1.6.2.2.2.1.4
18	(reserved)		
19	ER AAC LTP	ISO/IEC 14496-3 subpart 4	see subclause 1.6.2.2.2.1.4
20	ER AAC scalable	ISO/IEC 14496-3 subpart 4	see subclause 1.6.2.2.2.1.4
21	ER Twin VQ	ISO/IEC 14496-3 subpart 4	
22	ER BSAC	ISO/IEC 14496-3 subpart 4	
23	ER AAC LD	ISO/IEC 14496-3 subpart 4	see subclause 1.6.2.2.2.1.4
24	ER CELP	ISO/IEC 14496-3 subpart 3	
25	ER HVXC	ISO/IEC 14496-3 subpart 2	
26	ER HILN	ISO/IEC 14496-3 subpart 7	
27	ER Parametric	ISO/IEC 14496-3 subpart 2 and 7	
28	SSC	ISO/IEC 14496-3 subpart 8	
29	(reserved)		
30	(reserved)		
31	(escape)		
32	Layer-1	ISO/IEC 14496-3 subpart 9	
33	Layer-2	ISO/IEC 14496-3 subpart 9	
34	Layer-3	ISO/IEC 14496-3 subpart 9	
35	DST	ISO/IEC 14496-3 subpart 10	

1.6.2.2.2 Mapping of audio payloads to access units and elementary streams

1.6.2.2.2.1 AAC Main, AAC LC, AAC SSR, AAC LTP

One top level payload (raw_data_block()) is mapped into one access unit. Subsequent access units form one elementary stream.

1.6.2.2.2.2 AAC scalable

One top level payload (aac_scalable_main_element(), ASME) is mapped into one access unit. Subsequent access units form one elementary stream.

One top level payload (aac_scalable_extension_element(), ASEE) is mapped into one access unit. Subsequent access units of the same enhancement layer form one elementary stream. This results in individual elementary streams for each layer.

The streams of subsequent layers depend on each other.

1.6.2.2.2.3 ER AAC LC, ER AAC SSR, ER AAC LTP, ER AAC scalable, ER AAC LD

The data elements of the according top level payload (er_raw_data_block(), aac_scalable_main_element(), aac_scalable_extension_element()) are subdivided into different categories depending on its error sensitivity and collected in instances of these categories (see subpart 4, subclause 4.5.2.4). Depending on the value of epConfig, there are several ways to map these instances to access units to form one or several elementary streams (see subclause 1.6.3.6). Subsequent elementary streams depend on each other.

Note: The bits of the data function byte_alignment() terminating the AAC top level payloads may be omitted if the data elements of the according top level payload is not directly maped into one access unit. Hence, it might be ommitted if the top level payload is split (e.g. in case of epConfig=1) or post-processed (e.g. in case of epConfig=3).

1.6.3 Semantics

1.6.3.1 AudioObjectType

A five bit field indicating the audio object type. This is the master switch which selects the actual bitstream payload syntax of the audio data. In general, different object type use a different bitstream payload syntax. The interpretation of this field is given in Table 1.1.

1.6.3.2 AudioObjectTypeExt

This data element extends the range of audio object types.

1.6.3.3 samplingFrequency

The sampling frequency used for this audio object. Either transmitted directly, or coded in the form of **samplingFrequencyIndex.**

1.6.3.4 samplingFrequencyIndex

A four bit field indicating the sampling rate used. If samplingFrequencyIndex equals 15 then the actual sampling rate is signaled directly by the value of **samplingFrequency**. In all other cases **samplingFrequency** is set to the value of the corresponding entry in Table 1.16.

Table 1.16 — Sampling Frequency Index

samplingFrequencyIndex	Value
0x0	96000
0x1	88200
0x2	64000
0x3	48000
0x4	44100
0x5	32000
0x6	24000
0x7	22050
0x8	16000
0x9	12000
0xa	11025
0xb	8000
0xc	7350
0xd	reserved
0xe	reserved
0xf	escape value

1.6.3.5 channelConfiguration

A four bit field indicating the audio output channel configuration:

Table 1.17 — Channel Configuration

value	number of channels	audio syntactic elements, listed in order received	channel to speaker mapping
0	-	-	defined in GASpecificConfig
1	1	single_channel_element()	center front speaker
2	2	channel_pair_element()	left, right front speakers
3	3	single_channel_element(), channel_pair_element()	center front speaker, left, right front speakers
4	4	single_channel_element(), channel_pair_element(), single_channel_element()	center front speaker, left, right center front speakers, rear surround speakers
5	5	single_channel_element(), channel_pair_element(), channel_pair_element()	center front speaker, left, right front speakers, left surround, right surround rear speakers
6	5+1	single_channel_element(), channel_pair_element(), channel_pair_element(), lfe _element()	center front speaker, left, right front speakers, left surround, right surround rear speakers, front low frequency effects speaker
7	7+1	single_channel_element(), channel_pair_element(), channel_pair_element(), channel_pair_element(), lfe_element()	center front speaker left, right center front speakers, left, right outside front speakers, left surround, right surround rear speakers, front low frequency effects speaker
8-15	-	-	reserved

1.6.3.6 epConfig

This data element signals what kind of error robust configuration is used.

Table 1.18 — epConfig

epConfig	Description
0	All instances of all error sensitivity categories belonging to one frame are stored within one access unit. There exists one elementary stream per scalability layer, or just one elementary stream in case of non-scalable configurations.
1	Each instance of each sensitivity category belonging to one frame is stored separately within a single access unit, i.e. there exist as many elementary streams as instances defined within a frame.
2	The error protection decoder has to be applied. The definition of EP classes is not normatively defined, but defined at application level. However, the restictions imposed on EP classes as defined in subclause 1.7 must be satisfied.
3	The error protection decoder has to be applied. The mapping between EP classes and ESC instances is signaled by the data element directMapping.

1.6.3.7 direct mapping

This data element identifies the mapping between error protection classes and error sensitivity category instances.

Table 1.19 — directMapping

directMapping	Description
0	Reserved
1	Each error protection class is treated as an instance of an error sensitivity category (one to one mapping), so that the error protection decoder output is equivalent to epConfig=1 data.

1.6.3.8 extensionSamplingFrequencyIndex

A four bit field indicating the output sampling frequency of the extension tool corresponding to the extensionAudioObjectType, according to Table 1.16

1.6.3.9 extensionSamplingFrequency

The output sampling frequency of the extension tool corresponding to the extensionAudioObjectType. Either transmitted directly, or coded in the form of extensionSamplingFrequencyIndex.

1.6.3.10 bits_to_decode

A helper function; returns the number of bits not yet decoded in the current AudioSpecificConfig(), if the length of this element has been signaled by a system/transport layer. If the length of this element is unknown, bits_to_decode() returns 0.

1.6.3.11 syncExtensionType

Syncword which marks the beginning of appended extension configuration data**.** This configuration data corresponds to an extension tool of which the coded data is embedded (in a backward compatible manner) in that of the underlying audioObjectType. If syncExtensionType is present, the configuration data of the extension tool is separated from that of the underlying audioObjectType, which allows for backward compatible signaling (see subclause 1.6.5). Decoders that do not support the extension tool can ignore the extension tool configuration data. Note that this backward compatible signaling can only be used in MPEG-4 based systems that convey the length of the AudioSpecificConfig().

1.6.3.12 sbrPresentFlag

A flag indicating the presence or absence of SBR data in case of extensionAudioObjectType==5 (i.e. explicit SBR signaling, see subclause 1.6.5). The value –1 indicates that the sbrPresentFlag was not conveyed in the

AudioSpecificConfig(). In this case, a High Efficiency AAC Profile decoder shall be able to detect the presence of SBR data in the Elementary Stream (i.e. implicit SBR signaling, see subclause 1.6.5).

1.6.3.13 extensionAudioObjectType

A five bit field indicating the extension audio object type. This object type corresponds to an extension tool, which is used to enhance the underlying audioObjectType.

1.6.4 Upstream

1.6.4.1 Introduction

Upstreams are defined to allow for user on a remote side to dynamically control the streaming of the server.

The need for an up-stream channel is signaled to the client terminal by supplying an appropriate elementary stream descriptor declaring the parameters for that stream. The client terminal opens this up-stream channel in a similar manner as it opens the downstream channels. The entities (e.g. media encoders & decoders) that are connected through an up-stream channel are known from the parameters in its elementary stream descriptor and from the association of the elementary stream descriptor to a specific object descriptor.

An up-stream can be associated to a single downstream or a group of down streams. The stream type of the downstream to which the up-stream is associated defines the scope of the up-stream. When the up-stream is associated to a single downstream it carries messages about the downstream it is associated to. The syntax and semantics of messages for MPEG-4 Audio are defined in the next subclause.

1.6.4.2 Syntax

Table 1.20 — Syntax of AudioUpstreamPayload()

Syntax	No. of bits	Mnemonic
AudioUpstreamPayload()		
{		
upStreamType;	**4**	**uimsbf**
switch (upStreamType) {		
case 0: /* scalablity control */		
numOfLayer;	**6**	**uimsbf**
for (layer = 0; layer < numOfLayer; layer++) {		
avgBitrate[layer];	**24**	**uimsbf**
}		
break;		
case 1: /* BSAC frame interleaving */		
numOfSubFrame;	**5**	**uimsbf**
break;		
case 2: /* quality feedback */		
multiLayOrSynEle;	**1**	**uimsbf**
if (multiLayOrSynEle) {		
layOrSynEle;	**6**	**uimsbf**
}		
else {		
layOrSynEle = 1;		
}		
numFrameExp[layOrSynEle];	**4**	**uimsbf**
lostFrames[layOrSynEle];	**numFrameExp [layOrSynEle]**	**uimsbf**
break;		
case 3: /* bitrate control */		
avgBitrate;	**24**	**uimsbf**
break;		
default: /* reserved for future use */		
break;		

```
    }
}
```

1.6.4.3 Definitions

upStreamType A 4-bit unsigned integer value representing the type of the up-stream as defined in the following Table 1.21

Table 1.21 — Definition of upStreamType

UpStreamType	Type of Audio up-stream
0	scalability control
1	BSAC frame interleaving
2	quality feedback
3	bitrate control
4 – 15	reserved for future use

avgBitrate[layer] The average bitrate in bits per second of a large step layer, which the client requests to be transmitted from the server.

numOfSubFrame A 5-bit unsigned integer value representing the number of the frames which are grouped and transmitted in order to reduce the transmission overhead. The transmission overhead is decreased but the delay is increased as numOfSubFrame is increased.

multiLayOrSynEle This bit signals, whether or not a multi-channel or multi-layer configuration is used. Only in that case a layer number or a syntactic element number needs is transmitted.

layOrSynEle A 6-bit unsigned integer value representing the number of the syntactic elements (in case of multi-channel setup) or the number of the layers (in case of multi-layer setup), to which the following quality feedback information belongs. This number refers to one of the layers or one of the syntactic elements contained within the associated Audio object. If the Audio object does neither support scalability nor multi-channel capabilities, this value is implicitly set to 1.

numFrameExp[layOrSynEle] This value indicates the number of last recently passed frames $\left(2^{numFrameExp}-1\right)$ considered in the following lostFrames value.

lostFrames[layOrSynEle] This field contains the number of lost frames with respect to the indicated layer or syntactic element within the last recently passed frames signalled by numFrameExp.

avgBitrate The average bitrate in bits per second of the whole Audio object, which the client requests to be transmitted from the server.

1.6.4.4 Decoding process

First, **upStreamType** is parsed which represents the type of the up-stream. The remaining decoding process depends upon the type of the up-stream.

1.6.4.4.1 Decoding of scalability control

Next is the value **numOfLayer**. It represents the number of the data elements **avgBitrate** to be read. **avgBitrate** follows.

1.6.4.4.2 Decoding of BSAC frame interleaving

The data element to be read is **numOfSubFrame**. It represents the number of the sub-frames to be interleaved in BSAC tool. BSAC can allow for runtime adjustments to the quality of service. When the content of upstream is transmitted from the client to the server to implement a stream dynamically and interactively. BSAC data are split and interleaved in the server. The detailed process for implementing an AU payload in the server is described in Annex 4.B.17.

1.6.4.4.3 Decoding of quality feedback

The real frame loss rate in percent can be derived using the following formula:

$$frameLossRate[layOrSynEle] = \frac{lostFrames[layOrSynEle]}{2^{numFrameExp[layOrSynEle]} - 1} * 100\%$$

1.6.4.4.4 Decoding of bitrate control

avgBitrate is parsed**.**

1.6.5 Signaling of SBR

1.6.5.1 Generating and signaling AAC+SBR content

The SBR tool in combination with the AAC coder provides a significant increase of audio compression efficiency. At the same time it allows for compatibility with existing AAC-only decoders. However, the audio quality for decoders without the SBR tool will of course be significantly lower than for those supporting the SBR tool. Therefore, depending on the application, a content provider or content creator will want to choose between the two alternatives given below. In general, the SBR data is always embedded in the AAC stream in an AAC compatible way (in the extension_payload), and SBR is a pure post processing step in the decoder. Therefore, compatibility can be achieved. However, by means of different signaling the content creator can select between the full-quality mode and the backward compatibility mode as follows.

1.6.5.1.1 Ensuring full audio quality of AAC+SBR for the listener

To ensure that all listeners get the full audio quality of AAC+SBR, the stream generated shall only play on SBR capable decoders (decoders that support the HE AAC Profile, hereinafter referred to as HE AAC Profile decoders). This is achieved by indicating the HE AAC profile and using the explicit, hierarchical signalling (signaling 2.A. as described below). As a result, decoders without SBR support will not play such streams. With regard to AAC-only streams, an HE AAC Profile decoders will decode all AAC Profile streams of the appropriate level, as the HE AAC Profile is a superset of the AAC Profile.

1.6.5.1.2 Achieving backward compatibility with existing AAC-only decoders

The aim of this mode is to get all AAC-based decoders to play the stream, even if they don't support the SBR tool. Compatible streams can be created using the following two signaling methods:

a) indicating a profile containing AAC (e.g. the AAC Profile), except the HE AAC Profile, and using the explicit backward compatible signalling (2.B. as described below). This method is recommended for all MPEG-4 based systems in which the length of the AudioSpecificConfig() is known in the decoder. As this is not the case for LATM with audioMuxVersion==0 (see subclause 1.7), this method cannot be used for LATM with audioMuxVersion==0. In explicit backward compatible signaling, SBR-specific configuration data is added at the end of the AudioSpecificConfig(). Decoders that do not know about SBR will ignore these parts, while HE AAC Profile decoders will detect its presence and configure the decoder accordingly.

b) indicating a profile containing AAC (e.g. the AAC Profile, or an MPEG-2 AAC profile), except the HE AAC Profile, and using implicit signalling. In this mode, there is no explicit indication of the presence of SBR data. Instead, decoders check the presence while decoding the stream and use the SBR tool if SBR data is found. This is possible because SBR can be decoded without SBR-specific configuration data if a certain way of handling decoder output sample rate is obeyed, as described below for HE AAC Profile decoders.

Both methods lead to the result that the AAC part of an AAC+SBR streams will be decoded by AAC-only decoders. AAC+SBR decoders will detect the presence of SBR and decode the full quality AAC+SBR stream.

1.6.5.2 Implicit and explicit signaling of SBR

This subclause outlines the different signaling methods of SBR, and the decoder behavior for different types of signaling.

There are several ways to signal the presence of SBR data:

1. **implicit signaling:** If EXT_SBR_DATA or EXT_SBR_DATA_CRC extension_payload() elements are detected in the bitstream payload, this implicitly signals the presence of SBR data. The ability to detect and decode implicitly signaled SBR is mandatory for all High Efficiency AAC Profile (HE AAC Profile) decoders.

2. **explicit signaling:** The presence of SBR data is signaled explicitly by means of the SBR Audio Object Type in the AudioSpecificConfig(). When explicit signaling is used, implicit signaling shall not occur. Two different types of explicit signaling are available:

   2.A. **hierarchical signaling:** If the first audioObjectType (AOT) signaled is the SBR AOT, a second audio object type is signaled which indicates the underlying audio object type. This signaling method is not backward compatible.

   2.B. **backward compatible signaling:** The extensionAudioObjectType is signaled at the end of the AudioSpecificConfig(). This method shall only be used in systems that convey the length of the AudioSpecificConfig(). Hence, it shall not be used for LATM with audioMuxVersion==0.

Table 1.22 shows the decoder behavior depending on profile and audio object type indication when implicit or explicit signaling is used.

Table 1.22 — SBR Signaling and Corresponding Decoder Behavior

Bitstream payload characteristics				Decoder behavior (Note 4)	
Profile indication	**extension AudioObjectType**	**sbrPresent Flag**	**raw_data_block**	**AAC decoders not supporting HE AAC Profile**	**AAC decoders supporting HE AAC Profile**
Profiles with AAC support other than High Efficiency AAC Profile	!= SBR (signaling 1)	-1 (Note 1)	AAC	Play AAC	Play AAC
			AAC+SBR	Play AAC	Play at least AAC, should play AAC+SBR
	== SBR (signaling 2.B)	0 (Note 2)	AAC	Play AAC	Play AAC
		1 (Note 3)	AAC+SBR	Play AAC	Play at least AAC, should play AAC+SBR
High Efficiency AAC Profile	== SBR (signaling 2.A or 2.B)	1 (Note 3)	AAC+SBR	Unsupported Profile - Don't play	Play AAC+SBR

Note 1: Implicit signaling, check payload in order to determine output sampling frequency, or assume the presence of SBR data in the payload, giving an output sampling frequency of twice the sampling frequency indicated by samplingFrequency in the AudioSpecificConfig() (unless the down sampled SBR Tool is operated, or twice the sampling frequency indicated by samplingFrequency exceeds the maximum allowed output sampling frequency of the current level, in which case the output sampling frequency is the same as indicated by samplingFrequency).

Note 2: Explicitly signals that there is no SBR data, hence no implicit signaling is present, and the output sampling frequency is given by samplingFrequency in the AudioSpecificConfig().

Note 3: Output sampling frequency is the extensionSamplingFrequency in AudioSpecificConfig().

Note 4: In all cases a decoder has to support the Profile and Level indicated in the bitstream payload in order to be able to decode and play the content of the bitstream payload.

The upper part of Table 1.22 displays bitstream payload characteristics and decoder behavior if the profile indication is any profile with AAC, apart from the High Efficiency AAC Profile. The lower part displays bitstream payload characteristics and decoder behavior if the profile indication is the High Efficiency AAC Profile.

1.6.5.3 HE AAC profile decoder behavior in case of implicit signaling

If the presence of SBR data is backward compatible implicitly signaled (signaling 1 in the list above) the extensionAudioObjectType is not the SBR AOT, and the sbrPresentFlag is set to –1, indicating that implicit signaling may occur.

Since the HE AAC Profile decoder is a dual rate system, with the SBR Tool operating at twice the sample rate of the underlying AAC decoder, the output sample rate cannot be assumed to be that of the AAC decoder just because SBR is not explicitly signaled. The decoder shall determine the output sample rate by either of the following two methods:

- Check for the presence of SBR data in the bitstream payload prior to decoding. If no SBR data is found, the output sample rate is equal to that signaled as samplingFrequency in the AudioSpecificConfig(). If SBR data is found the output sample rate is twice that signaled as samplingFrequency in the AudioSpecificConfig

- Assume that the SBR data is available and decide the output sample rate to be twice that signaled in the AudioSpecificConfig(). If no SBR data is found once the decoding process has started, the SBR Tool can be used for upsampling only, as described in subclause 4.6.18.5.

The above only applies if twice the sample rate signaled in the AudioSpecificConfig() does not exceed the maximum output sample rate allowed for the current level. Hence, for a HE AAC Profile decoder of levels 2, 3, or 4, the output sample rate is equal to the sample rate signaled in the AudioSpecificConfig() if the latter exceeds 24kHz.

The down sampled SBR Tool shall be used when needed to ensure that the output sample rate does not exceed the maximum allowed sample rate of the present level of the High Efficiency AAC Profile decoder.

1.6.5.4 HE AAC profile decoder behavior in case of explicit signaling

If the presence of SBR data is explicitly signaled (signaling 2, in the list above) the presence of SBR data is backward compatible explicitly signaled (signaling 2.B) or non-backward explicitly signaled (signaling 2.A).

For the backward compatible explicit signaling (signaling 2.B) the extensionAudioObjectType signaled is the SBR AOT. For this backward compatible explicit signaling the sbrPresentFlag is transmitted and can be either zero or one. If the sbrPresentFlag is zero, this indicates that SBR data is not present, and hence the HE AAC Profile decoder does not have to check the extension_payload() for the presence of SBR data or make assumptions on the output sample rate in anticipation of SBR data. If the sbrPresentFlag is one, SBR data is present and the HE AAC Profile decoder shall operate the SBR Tool.

For the non-backward compatible explicit signaling of SBR (signaling 2.A) the extensionAudioObjectType signaled is the SBR AOT. For this hierarchical explicit signaling, the sbrPresentFlag is set to one if the extensionAudioObjectType is SBR. The sbrPresentFlag is not transmitted and hence it is not possible to explicitly signal the absence of implicit signaling. Hence, for the hierarchical explicit signaling, SBR data is always present and the HE AAC Profile decoder shall operate the SBR Tool.

The down sampled SBR Tool shall be operated if the output sample rate would otherwise exceed the maximum allowed output sample rate for the present level, or if the extensionSamplingFrequency is the same as the samplingFrequency.

1.7 MPEG-4 Audio transport stream

1.7.1 Overview

This subclause defines a mechanism to transport ISO/IEC 14496-3 (MPEG-4 Audio) streams without using ISO/IEC 14496-1 (MPEG-4 Systems) for audio-only applications. Figure 1.1 shows the concept of MPEG-4 Audio transport. The transport mechanism uses a two-layer approach, namely a multiplex layer and a synchronization layer. The multiplex layer (Low-overhead MPEG-4 Audio Transport Multiplex: LATM) manages multiplexing of several MPEG-4 Audio payloads and their AudioSpecificConfig() elements. The synchronization layer specifies a self-synchronized syntax of the MPEG-4 Audio transport stream which is called Low Overhead Audio Stream (LOAS). The interface format to a transmission layer depends on the conditions of the underlying transmission layer as follows:

- LOAS shall be used for the transmission over channels where no frame synchronization is available.
- LOAS may be used for the transmission over channels with fixed frame synchronization.
- A multiplexed element (AudioMuxElement() / EPMuxElement()) without synchronization shall be used only for transmission channels where an underlying transport layer already provides frame synchronization that can handle arbitrary frame size.

The details of the LOAS and the LATM formats are described in subclauses 1.7.2 and 1.7.3, respectively.

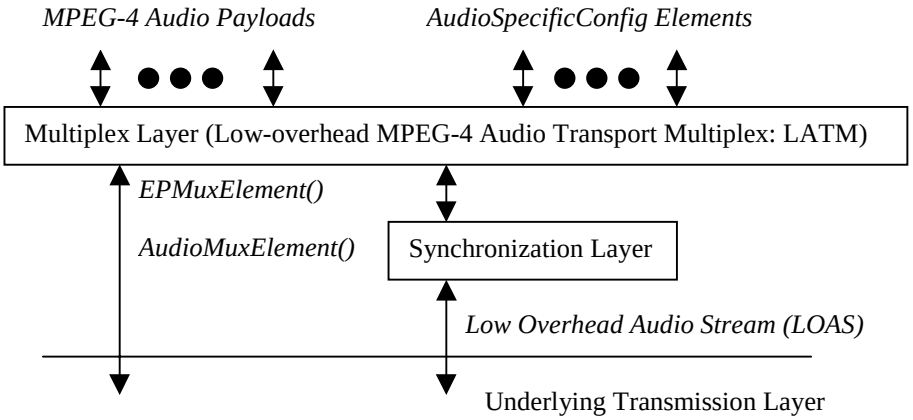


Figure 1.1 — Concept of MPEG-4 Audio Transport

The mechanism defined in this subclause should not be used for transmission of TTSI objects (12), Main Synthetic objects (13), Wavetable Synthesis objects (14), General MIDI objects (15) and Algorithmic Synthesis and Audio FX objects (16). It should further not be used for transmission of any object in conjunction with (epConfig==1). For those objects, other multiplex and transport mechanisms might be used, e.g. those defined in MPEG-4 Systems.

1.7.2 Synchronization Layer

The synchronization layer provides the multiplexed element with a self-synchronized mechanism to generate LOAS. The LOAS has three different types of format, namely AudioSyncStream(), EPAudioSyncStream() and AudioPointerStream(). The choice for one of the three formats is dependent on the underlying transmission layer.

- AudioSyncStream()

AudioSyncStream() consists of a syncword, the multiplexed element with byte alignment, and its length information. The maximum byte-distance between two syncwords is 8192 bytes. This self-synchronized stream shall be used for the case that the underlying transmission layer comes without any frame synchronization.

- EPAudioSyncStream()

For error prone channels, an alternative version to AudioSyncStream() is provided. This format has the same basic functionality as the previously described AudioSyncStream(). However, it additionally provides a longer syncword and a frame counter to detect lost frames. The length information and the frame counter are additionally protected by a FEC code.

- AudioPointerStream()

AudioPointerStream() shall be used for applications using an underlying transmission layer with fixed frame synchronization, where transmission framing cannot be synchronized with the variable length multiplexed element. Figure 1.2 shows synchronization in AudioPointerStream(). This format utilizes a pointer indicating the start of the next multiplex element in order to synchronize the variable length payload with the constant transmission frame.

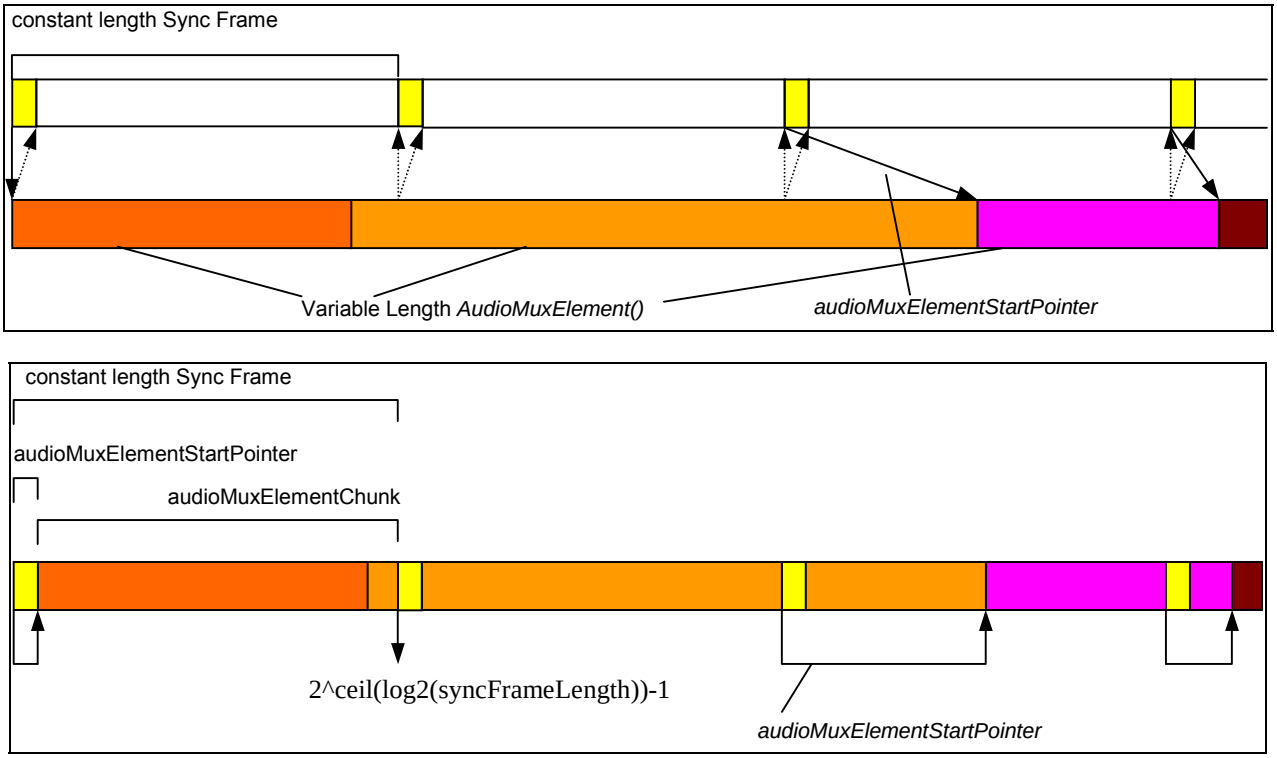


Figure 1.2 — Synchronization in AudioPointerStream()

1.7.2.1 Syntax

Table 1.23 — Syntax of AudioSyncStream()

Syntax	No. of bits	Mnemonic
AudioSyncStream()		
{		
while (nextbits() == **0x2B7**) { /* syncword */	**11**	**bslbf**
audioMuxLengthBytes;	**13**	**uimsbf**
AudioMuxElement(1);		
}		
}		

Table 1.24 — Syntax of EPAudioSyncStream()

Syntax	No. of bits	Mnemonic
EPAudioSyncStream()		
{		
while (nextbits() == **0x4de1**) { /* syncword */	**16**	**bslbf**
futureUse;	**4**	**uimsbf**
audioMuxLengthBytes;	**13**	**uimsbf**
frameCounter;	**5**	**uimsbf**
headerParity;	**18**	**bslbf**
EPMuxElement(1, 1);		
}		
}		

Table 1.25 — Syntax of AudioPointerStream()

Syntax	No. of bits	Mnemonic

AudioPointerStream (syncFrameLength)
{
  while (! EndOfStream) {
  AudioPointerStreamFrame (syncFrameLength);
  }
}		

Table 1.26 — Syntax of AudioPointerStreamFrame()

Syntax	No. of bits	Mnemonic
AudioPointerStreamFrame(length) {		
audioMuxElementStartPointer;	**ceil(log2(length))**	**uimsbf**
audioMuxElementChunk;	**length – ceil(log2(length))**	**bslbf**
}		

1.7.2.2 Semantics

1.7.2.2.1 AudioSyncStream()

audioMuxLengthBytes A 13-bit data element indicating the byte length of the subsequent AudioMuxElement() with byte alignment (AudioSyncStream) or the subsequent EPMuxElement() (EPAudioSyncStream).

AudioMuxElement() A multiplexed element as specified in subclause 1.7.3.2.2.

1.7.2.2.2 EPAudioSyncStream()

futureUse A 4-bit data element for future use, which shall be set to '0000'.

audioMuxLengthBytes see subclause 1.7.2.2.1.

frameCounter A 5-bit data element indicating a sequential number which is used to detect lost frames. The number is continuously incremented for each multiplexed element as a modulo counter.

headerParity A 18-bit data element which contains a BCH (36,18) code shortened from BCH (63,45) code for the elements **audioMuxLengthBytes** and **frameCounter**. The generator polynomial is $x^{18}+x^{17}+x^{16}+x^{15}+x^{9}+x^{7}+x^{6}+x^{3}+x^{2}+x+1$. The value is calculated with this generator polynomial as described in subclause 1.8.4.3.

EPMuxElement() An error resilient multiplexed element as specified in subclause 1.7.3.2.1.

1.7.2.2.3 AudioPointerStream()

AudioPointerStreamFrame() A sync frame of fixed length provided by an underlying transmission layer.

audioMuxElementStartPointer A data element indicating the starting point of the first AudioMuxElement() within the current AudioPointerStreamFrame(). The number of bits required for this data element is calculated as ceil(log2(syncFrameLength)). The transmission frame length has to be provided from the underlying transmission layer. The maximum possible value of this data element is reserved to signal that there is no start of an AudioMuxElement() in this sync frame.

audioMuxElementChunk A part of a concatenation of subsequent AudioMuxElement()'s (see Figure 1.2).

1.7.3 Multiplex Layer

The LATM layer multiplexes several MPEG-4 Audio payloads and AudioSpecificConfig() syntax elements into one multiplexed element. The multiplexed element format is selected between AudioMuxElement() and EPMuxElement() depending on whether error resilience is required in the multiplexed element itself, or not. EPMuxElement() is an error resilient version of AudioMuxElement() and may be used for error prone channels.

The multiplexed elements can be directly conveyed on transmission layers with frame synchronization. In this case, the first bit of the multiplexed element shall be located at the first bit of a transmission payload in the underlying transmission layer. If the transmission payload allows only byte-aligned payload, padding bits for byte alignment shall follow the multiplexed element. The number of the padding bits should be less than 8. These padding bits should be removed when the multiplexed element is de-multiplexed into the MPEG-4 Audio payloads. Then, the MPEG-4 Audio payloads are forwarded to the corresponding MPEG-4 Audio decoder tool.

Usage of LATM in case of scalable configurations with CELP core and AAC enhancement layer(s):

- *Instances of the AudioMuxElement() are transmitted in equidistant manner.*
- *The represented timeframe of one AudioMuxElement() is similar to a multiple of a super-frame timeframe.*
- *The relative number of bits for a certain layer within any AudioMuxElement() compared to the total number of bits within this AudioMuxElement() is equal to the relative bitrate of that layer compared to the bitrate of all layers.*
- *In case of coreFrameOffset = 0 and latmBufferFullness = 0, all core coder frames and all AAC frames of a certain super-frame are stored within the same instance of AudioMuxElement().*
- *In case of coreFrameOffset > 0, several or all core coder frames are stored within previous instances of AudioMuxElement().*
- *Any core layer related configuration information refers to the core frames transmitted within the current instance of the AudioMuxElement(), independent of the value of coreFrameOffset.*
- *A specified latmBufferFullness is related to the first AAC frame of the first super-frame stored within the current AudioMuxElement().*
- *The value of latmBufferFullness can be used to* **determine the location of the first bit of the first AAC frame of the current layer of the first super-frame stored within the current AudioMuxElement() by means of a backpointer***:*

  $$backPointer = -meanFrameLength + latmBufferFullness + currentFrameLength$$

  The backpointer value specifies the location as a negative offset from the current AudioMuxElement(), i. e. it points backwards to the beginning of an AAC frame located in already received data. Any data not belonging to the payload of the current AAC layer is not taken into account. If (latmBufferFullness == '0'), then the AAC frame starts after the current AudioMuxElement().

Note that the possible LATM configurations are restricted due to limited signalling capabilities of certain data elements as follows:

- Number of layers: 8 (numLayer has 3 bit)
- Number of streams: 16 (streamIndx has 4 bit)
- Number of chunks: 16 (numChunk has 4 bit)

1.7.3.1 Syntax

Table 1.27 — Syntax of EPMuxElement()

Syntax	No. of bits	Mnemonic
EPMuxElement(epDataPresent, muxConfigPresent)		
{		
if (epDataPresent) {		
epUsePreviousMuxConfig;	**1**	**bslbf**
epUsePreviousMuxConfigParity;	**2**	**bslbf**
if (!epUsePreviousMuxConfig) {		
epSpecificConfigLength;	**10**	**bslbf**
epSpecificConfigLengthParity;	**11**	**bslbf**
ErrorProtectionSpecificConfig();		
ErrorProtectionSpecificConfigParity();		
}		
ByteAlign();		
EPAudioMuxElement(muxConfigPresent);		
}		
else {		
AudioMuxElement(muxConfigPresent);		
}		
}		

Table 1.28 — Syntax of AudioMuxElement()

Syntax	No. of bits	Mnemonic
AudioMuxElement(muxConfigPresent)		
{		
if (muxConfigPresent) {		
useSameStreamMux;	**1**	**bslbf**
if (!useSameStreamMux)		
StreamMuxConfig();		
}		
if (audioMuxVersionA == 0) {		
for (i = 0; i <= numSubFrames; i++) {		
PayloadLengthInfo();		
PayloadMux();		
}		
if (otherDataPresent) {		
for(i = 0; i < otherDataLenBits; I++) {		
otherDataBit;	**1**	**bslbf**
}		
}		
}		
else {		
/* tbd */		
}		
ByteAlign();		
}		

Table 1.29 – Syntax of StreamMuxConfig()

Syntax	No. of bits	Mnemonic
StreamMuxConfig()		
{		
audioMuxVersion;	**1**	**bslbf**
if (audioMuxVersion == 1) {		
audioMuxVersionA;	**1**	**bslbf**
}		

```
else {
    audioMuxVersionA = 0;
}
if (audioMuxVersionA == 0) {
    if (audioMuxVersion == 1 ) {
        taraBufferFullness = LatmGetValue();
    }
    streamCnt = 0;
    allStreamsSameTimeFraming;                                  1       uimsbf
    numSubFrames;                                               6       uimsbf
    numProgram;                                                 4       uimsbf
    for (prog = 0; prog <= numProgram; prog++) {
        numLayer;                                               3       uimsbf
        for (lay = 0; lay <= numLayer; lay++) {
            progSIndx[streamCnt] = prog; laySIndx[streamCnt] = lay;
            streamID [ prog][ lay] = streamCnt++;
            if (prog == 0 && lay == 0) {
                useSameConfig = 0;
            } else {
                useSameConfig;                                  1       uimsbf
            }
            if (! useSameConfig) {
                if ( audioMuxVersion == 0 ) {
                    AudioSpecificConfig();
                }
                else {
                    ascLen = LatmGetValue();
                    ascLen -= AudioSpecificConfig();                    Note 1
                    fillBits;                                   ascLen  bslbf
                }
            }
            frameLengthType[streamID[prog][ lay]];              3       uimsbf
            if (frameLengthType[streamID[prog][lay] == 0) {
                latmBufferFullness[streamID[prog][ lay]];       8       uimsbf
                if (! allStreamsSameTimeFraming) {
                    if ((AudioObjectType[lay] == 6 ||
                        AudioObjectType[lay] == 20) &&
                        (AudioObjectType[lay-1] == 8 ||
                        AudioObjectType[lay-1] == 24)) {
                            coreFrameOffset;                    6       uimsbf

                    }
                }
            } else if (frameLengthType[streamID[prog][ lay]] == 1) {
                frameLength[streamID[prog][lay]];               9       uimsbf
            } else if (     frameLengthType[streamID[prog][ lay]] == 4 ||
                            frameLengthType[streamID[prog][ lay]] == 5 ||
                            frameLengthType[streamID[prog][ lay]] == 3 ) {
                CELPframeLengthTableIndex[streamID[prog][lay]];  6      uimsbf
            } else if (     frameLengthType[streamID[prog][ lay]] == 6 ||
                            frameLengthType[streamID[prog][ lay]] == 7 ) {
                HVXCframeLengthTableIndex[streamID[prog][ lay]]; 1      uimsbf
            }
        }
    }

    otherDataPresent;                                           1       uimsbf
    if (otherDataPresent) {
        if ( audioMuxVersion == 1 ) {
```

otherDataLenBits = LatmGetValue();		
}		
else {		
otherDataLenBits = 0; /* helper variable 32bit */		
do {		
otherDataLenBits *= 2^8;		
otherDataLenEsc;	**1**	**uimsbf**
otherDataLenTmp;	**8**	**uimsbf**
otherDataLenBits += otherDataLenTmp;		
} while (otherDataLenEsc);		
}		
}		
crcCheckPresent;	**1**	**uimsbf**
if (crcCheckPresent) **crcCheckSum;**	**8**	**uimsbf**
}		
else {		
/* tbd */		
}		
}		
Note 1: AudioSpecificConfig() returns the number of bits read.		

Table 1.30 — Syntax of LatmGetValue()

Syntax	No. of bits	Mnemonic
LatmGetValue()		
{		
bytesForValue;	**2**	**uimsbf**
value = 0; /* helper variable 32bit */		
for (i = 0; i <= bytesForValue; i++) {		
value *= 2^8;		
valueTmp;	**8**	**uimsbf**
value += valueTmp;		
}		
return value;		
}		

Table 1.31 — Syntax of PayloadLengthInfo()

Syntax	No. of bits	Mnemonic
PayloadLengthInfo()		
{		
if (allStreamsSameTimeFraming) {		
for (prog = 0; prog <= numProgram; prog++) {		
for (lay = 0; lay <= numLayer; lay++) {		
if (frameLengthType[streamID[prog][lay]] == 0) {		
MuxSlotLengthBytes[streamID[prog][lay]] = 0;		
do { /* always one complete access unit */		
tmp;	**8**	**uimsbf**
MuxSlotLengthBytes[streamID[prog][lay]] += tmp;		
} while(tmp == 255);		
} else {		
if (frameLengthType[streamID[prog][lay]] == 5 ||		
frameLengthType[streamID[prog][lay]] == 7 ||		
frameLengthType[streamID[prog][lay]] == 3) {		
MuxSlotLengthCoded[streamID[prog][lay]];	**2**	**uimsbf**
}		
}		
}		

```
        }
    } else {
        numChunk;                                                    4        uimsbf
        for (chunkCnt = 0; chunkCnt <= numChunk; chunkCnt++) {
            streamIndx;                                              4        uimsbf
            prog = progCIndx[chunkCnt] = progSIndx[streamIndx];
             lay   = layCIndx[chunkCnt]   = laySIndx   [streamIndx];
            if (frameLengthType[streamID[prog][lay]] == 0) {
                MuxSlotLengthBytes[streamID[prog][ lay]] = 0;
                do {  /* not necessarily a complete access unit */
                    tmp;                                             8        uimsbf
                    MuxSlotLengthBytes[streamID[prog][lay]] += tmp;
                } while (tmp == 255);
                AuEndFlag[streamID[prog][lay]];                      1        bslbf
            } else  {
                if (frameLengthType[streamID[prog][lay]] == 5 ||
                    frameLengthType[streamID[prog][lay]] == 7 ||
                    frameLengthType[streamID[prog][lay]] == 3) {
                    MuxSlotLengthCoded[streamID[prog][lay]];         2        uimsbf
                }
            }
        }
    }
}
```

Table 1.32 — Syntax of PayloadMux()

Syntax	No. of bits	Mnemonic
PayloadMux()		
{		
if (allStreamsSameTimeFraming) {		
for (prog = 0; prog <= numProgram; prog++) {		
for (lay = 0; lay <= numLayer; lay++) {		
payload [streamID[prog][lay]];		
}		
}		
} else {		
for (chunkCnt = 0; chunkCnt <= numChunk; chunkCnt++) {		
prog = progCIndx[chunkCnt];		
lay = layCIndx [chunkCnt];		
payload [streamID[prog][lay]];		
}		
}		
}		

1.7.3.2 Semantics

1.7.3.2.1 EPMuxElement()

For parsing of EPMuxElement(), an epDataPresent flag shall be additionally set at the underlying layer. If epDataPresent is set to 1, this indicates EPMuxElement() has error resiliency. If not, the format of EPMuxElement() is identical to AudioMuxElement(). The default for both flags is 1.

epDataPresent	Description
0	EPMuxElement() is identical to AudioMuxElement()
1	EPMuxElement() has error resiliency

epUsePreviousMuxConfig A flag indicating whether the configuration for the MPEG-4 Audio EP tool in the previous frame is applied in the current frame.

epUsePreviousMuxConfig	Description
0	The configuration for the MPEG-4 Audio EP tool is present.
1	The configuration for the MPEG-4 Audio EP tool is not present. The previous configuration should be applied.

epUsePreviousMuxConfigParity A 2-bits element which contains the parity for **epUsePreviousMuxConfig.** Each bit is a repetition of **epUsePreviousMuxConfig.** Majority decides.

epSpecificConfigLength A 10-bit data element to indicate the size of ErrorProtectionSpecificConfig()

epSpecificConfigLengthParity An 11-bit data element for epHeaderLength, calculated as described in subclause 1.8.4.3 with "1) Basic set of FEC codes".
Note: This means shortened Golay(23,12) is used

ErrorProtectionSpecificConfig() A data function covering configuration information for the EP tool which is applied to AudioMuxElement() as defined in subclause 1.8.2.1.

ErrorProtectionSpecificConfigParity() A data function covering the parity bits for **ErrorProtectionSpecificConfig()**, calculated as described in subclause 1.8.4.3, Table 1.46.

EPAudioMuxElement() A data function covering **e**rror resilient multiplexed element that is generated by applying the EP tool to AudioMuxElement() as specified by ErrorProtectionSpecificConfig(). Therefore data elements in AudioMuxElement() are subdivided into different categories depending on their error sensitivity and collected in instances of these categories. Following sensitivity categories are defined:

elements	error sensitivity category
useSameStreamMux + StreamMuxConfig()	0
PayloadLengthInfo()	1
PayloadMux()	2
otherDataBits	3

Note 1: There might be more than one instance of error sensitivity category 1 and 2 depending on the value of the variable **numSubFrames** defined in **StreamMuxConfig()**. Figure 1.3 shows an example for the order of the instances assuming numSubFrames is one (1).

Note 2: EPAudioMuxElement() has to be byte aligned, therefore **bit_stuffing** in ErrorProtectionSpecificConfig() should be always on.

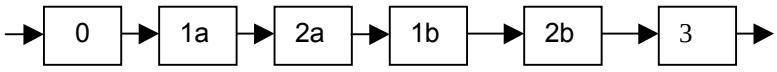


Figure 1.3 — Instance order in EPAudioMuxElement()

1.7.3.2.2 AudioMuxElement()

In order to parse an AudioMuxElement(), a muxConfigPresent flag shall be set at the underlying layer. If muxConfigPresent is set to 1, this indicates multiplexing configuration (StreamMuxConfig()) is multiplexed into

AudioMuxElement(), i.e. in-band transmission. If not, StreamMuxConfig() should be conveyed through out-band means, such as session announcement/description/control protocols.

muxConfigPresent	Description
0	out-band transmission of StreamMuxConfig()
1	in-band transmission of StreamMuxConfig()

useSameStreamMux A flag indicating whether the multiplexing configuration in the previous frame is applied in the current frame.

useSameStreamMux	Description
0	The multiplexing configuration is present.
1	The multiplexing configuration is not present. The previous configuration should be applied.

otherDataBit A 1-bit data element indicating the other data information.

1.7.3.2.3 StreamMuxConfig()

AudioSpecificConfig() is specified in subclause 1.6.2.1. In this case it constitutes a standalone element in itself (i.e. it does not extend the class BaseDescriptor as in the case of subclause 1.6).

audioMuxVersion A data element to signal the used multiplex syntax.
Note: In addition to (audioMuxVersion == 0), (audioMuxVersion == 1) supports the transmission of a taraBufferFullness and the transmission of the lengths of individual AudioSpecificConfig() data functions.

audioMuxVersionA A data element to signal the used syntax version. Possible values: 0 (default), 1 (reserved for future extensions).

taraBufferFullness A helper variable indicating the state of the bit reservoir in the course of encoding the LATM status information. It is transmitted as the number of available bits in the tara bit reservoir divided by 32 and truncated to an integer value. The maximum value that can be signaled using any setting of bytesForValue signals that the particular program and layer is of variable rate. This might be the value of hexadecimal FF (bytesForValue == 0), FFFF (bytesForValue == 1), FFFFFF (bytesForValue == 2) or FFFFFFFF (bytesForValue == 3). In these cases, buffer fullness is not applicable. The state of the bit reservoir is derived according to what is stated in subpart 4, subclause 4.5.3.2 (Bit reservoir). The LATM status information considered by the taraBufferFullness comprises any data of the AudioMuxElement() except of PayloadMux().

allStreamsSameTimeFraming A data element indicating whether all payloads, which are multiplexed in PayloadMux(), share a common time base.

numSubFrames A data element indicating how many PayloadMux() frames are multiplexed (numSubFrames+1). If more than one PayloadMux() frame is multiplexed, all PayloadMux() share a common StreamMuxConfig().The minimum value is 0 indicating 1 subframe.

numProgram A data element indicating how many programs are multiplexed (numProgram+1). The minimum value is 0 indicating 1 program.

numLayer A data element indicating how many scalable layers are multiplexed (numLayer+1). The minimum value is 0 indicating 1 layer.

useSameConfig A data element indicating that no AudioSpecificConfig() is transmitted but that the AudioSpecificConfig() most recently transmitted shall be applied.

useSameConfig	Description
0	AudioSpecificConfig() is present.
1	AudioSpecificConfig() is not present. AudioSpecificConfig() in the previous layer or program should be applied.

ascLen[prog][lay] A helper variable indicating the length in bits of the subsequent AudioSpecificConfig() data function including possible fill bits.

fillBits Fill bits.

frameLengthType A data element indicating the frame length type of the payload. For CELP and HVXC objects, the frame length (bits/frame) is stored in tables and only the indices to point out the frame length of the current payload is transmitted instead of sending the frame length value directly.

frameLengthType	Description
0	Payload with variable frame length. The payload length in bytes is directly specified with 8-bit codes in PayloadLengthInfo().
1	Payload with fixed frame length. The payload length in bits is specified with frameLength in StreamMuxConfig().
2	Reserved
3	Payload for a CELP object with one of 2 kinds of frame length. The payload length is specified by two table-indices, namely CELPframeLengthTableIndex and MuxSlotLengthCoded.
4	Payload for a CELP or ER_CELP object with fixed frame length. CELPframeLengthTableIndex specifies the payload length.
5	Payload for an ER_CELP object with one of 4 kinds of frame length. The payload length is specified by two table-indices, namely CELPframeLengthTableIndex and MuxSlotLengthCoded.
6	Payload for a HVXC or ER_HVXC object with fixed frame length. HVXCframeLengthTableIndex specifies the payload length.
7	Payload for an HVXC or ER_HVXC object with one of 4 kinds of frame length. The payload length is specified by two table-indices, namely HVXCframeLengthTableIndex and MuxSlotLengthCoded.

latmBufferFullness[streamID[prog][lay]] data element indicating the state of the bit reservoir in the course of encoding the first access unit of a particular program and layer in an AudioMuxElement(). It is transmitted as the number of available bits in the bit reservoir divided by the NCC divided by 32 and truncated to an integer value. A value of hexadecimal FF signals that the particular program and layer is of variable rate. In this case, buffer fullness is not applicable. The state of the bit reservoir is derived according to what is stated in subpart 4, subclause 4.5.3.2 (Bit reservoir). In the case of (audioMuxVersion == 0), bits spend for data other than any payload (e.g. multiplex status information or other data) are considered in the first occuring latmBufferFullness in an AudioMuxElement(). For AAC, the limitations given by the minimum decoder input buffer apply (see subpart 4, subclause 4.5.3.1). In the case of (allStreamsSameTimeFraming==1), and if only one program and one layer is present, this leads to an LATM configuration similar to ADTS.
In the case of (audioMUxVersion == 1), bits spend for data other than any payload are considered by taraBufferFullness.

coreFrameOffset identifies the first CELP frame of the current super-frame. It is defined only in case of scalable configurations with CELP core and AAC enhancement layer(s) and transmitted with the first AAC enhancement layer. The value 0 identifies the first CELP frame following StreamMuxConfig() as the first CELP frame of the current super-frame. A value > 0 signals the number of CELP frames that the first CELP frame of the current super-frame is transmitted earlier.

frameLength A data element indicating the frame length of the payload with frameLengthType of 1. The payload length in bits is specified as 8 * (frameLength + 20).

CELPframeLengthTableIndex A data element indicating one of two indices for pointing out the frame length for a CELP or ER_CELP object. (Table 1.34 and Table 1.35)

HVXCframeLengthTableIndex A data element indicating one of two indices for pointing out the frame length for a HVXC or ER_HVXC object. (Table 1.33)

otherDataPresent A flag indicating the presence of the other data than audio payloads.

otherDataPresent	Description
0	The other data than audio payload otherData is not multiplexed.
1	The other data than audio payload otherData is multiplexed.

otherDataLenBits A helper variable indicating the length in bits of the other data.

crcCheckPresent A data element indicating the presence of CRC check bits for the StreamMuxConfig() data functions.

crcCheckPresent	Description
0	CRC check bits are not present.
1	CRC check bits are present.

crcCheckSum CRC error detection data. This CRC uses the generation polynomial CRC8, as defined in subclause 1.8.4.5 and covers the entire StreamMuxConfig() upto but excluding the crcCheckPresent bit.

1.7.3.2.4 LatmGetValue()

bytesForValue A data element indicating the number of occurencies of the data element valueTmp.

valueTmp A data element used to calculate the helper variable value.

value A helper variable representing a value returned by the data function LatmGetValue().

1.7.3.2.5 PayloadLengthInfo()

tmp A data element indicating the payload length of the payload with frameLengthType of 0. The value 255 is used as an escape value and indicates that at least one more **tmp** value is following. The overall length of the transmitted payload is calculated by summing up the partial values.

MuxSlotLengthCoded A data element indicating one of two indices for pointing out the payload length for CELP, HVXC, ER_CELP, and ER_HVXC objects.

numChunk A data element indicating the number of payload chunks (numChunk+1). Each chunk may belong to an access unit with a different time base; only used if

allStreamsSameTimeFraming is set to zero. The minimum value is 0 indicating 1 chunk.

streamIndx A data element indicating the stream. Used if payloads are splitted into chunks.

chunkCnt Helper variable to count number of chunks.

progSIndx,laySIndx Helper variables to identify program and layer number from **streamIndx.**

progCIndx,layCIndx Helper variables to identify program and layer number from **chunkCnt.**

AuEndFlag A flag indicating whether the payload is the last fragment, in the case that an access unit is transmitted in pieces.

AuEndFlag	Description
0	The fragmented piece is not the last one.
1	The fragmented piece is the last one.

1.7.3.2.6 PayloadMux()

payload The actual audio payload by means of either an access unit (allStreamsSameTimeFraming == 1) or a part of a concatenation of subsequent access units (allStreamsSameTimeFraming == 0).

1.7.3.3 Tables

Table 1.33 — Frame length of HVXC [bits]

		MuxSlotLengthCoded			
frameLengthType[]	HVXCframeLengthTableIndex[]	00	01	10	11
6	0	40			
6	1	80			
7	0	40	28	2	0
7	1	80	40	25	3

Table 1.34 — Frame Length of CELP Layer 0 [bits]

	Fixed-Rate frameLengthType[] =4	1-of-4 Rates (Silence Compression) frameLengthType[]=5				1-of-2 Rates (FRC) frameLengthType[]=3	
		MuxSlotLengthCoded				MuxSlotLengthCoded	
CELPframeLenghTable Index		00	01	10	11	00	01
0	154	156	23	8	2	156	134
1	170	172	23	8	2	172	150
2	186	188	23	8	2	188	166
3	147	149	23	8	2	149	127
4	156	158	23	8	2	158	136
5	165	167	23	8	2	167	145
6	114	116	23	8	2	116	94
7	120	122	23	8	2	122	100
8	126	128	23	8	2	128	106
9	132	134	23	8	2	134	112
10	138	140	23	8	2	140	118
11	142	144	23	8	2	144	122
12	146	148	23	8	2	148	126
13	154	156	23	8	2	156	134

14	166	168	23	8	2	168	146
15	174	176	23	8	2	176	154
16	182	184	23	8	2	184	162
17	190	192	23	8	2	192	170
18	198	200	23	8	2	200	178
19	206	208	23	8	2	208	186
20	210	212	23	8	2	212	190
21	214	216	23	8	2	216	194
22	110	112	23	8	2	112	90
23	114	116	23	8	2	116	94
24	118	120	23	8	2	120	98
25	120	122	23	8	2	122	100
26	122	124	23	8	2	124	102
27	186	188	23	8	2	188	166
28	218	220	40	8	2	220	174
29	230	232	40	8	2	232	186
30	242	244	40	8	2	244	198
31	254	256	40	8	2	256	210
32	266	268	40	8	2	268	222
33	278	280	40	8	2	280	234
34	286	288	40	8	2	288	242
35	294	296	40	8	2	296	250
36	318	320	40	8	2	320	276
37	342	344	40	8	2	344	298
38	358	360	40	8	2	360	314
39	374	376	40	8	2	376	330
40	390	392	40	8	2	392	346
41	406	408	40	8	2	408	362
42	422	424	40	8	2	424	378
43	136	138	40	8	2	138	92
44	142	144	40	8	2	144	98
45	148	150	40	8	2	150	104
46	154	156	40	8	2	156	110
47	160	162	40	8	2	162	116
48	166	168	40	8	2	168	122
49	170	172	40	8	2	172	126
50	174	176	40	8	2	176	130
51	186	188	40	8	2	188	142
52	198	200	40	8	2	200	154
53	206	208	40	8	2	208	162
54	214	216	40	8	2	216	170
55	222	224	40	8	2	224	178
56	230	232	40	8	2	232	186
57	238	240	40	8	2	240	194
58	216	218	40	8	2	218	172
59	160	162	40	8	2	162	116
60	280	282	40	8	2	282	238
61	338	340	40	8	2	340	296
62-63	reserved						

Table 1.35 — Frame Length of CELP Layer 1-5 [bits]

	Fixed-Rate frameLengthType []=4	1-of-4 Rates (Silence Compression) frameLengthType[]=5			
		MuxSlotLengthCoded			
CELPframeLenghTableIndex		00	01	10	11
0	80	80	0	0	0

1	60	60	0	0	0
2	40	40	0	0	0
3	20	20	0	0	0
4	368	368	21	0	0
5	416	416	21	0	0
6	464	464	21	0	0
7	496	496	21	0	0
8	284	284	21	0	0
9	320	320	21	0	0
10	356	356	21	0	0
11	380	380	21	0	0
12	200	200	21	0	0
13	224	224	21	0	0
14	248	248	21	0	0
15	264	264	21	0	0
16	116	116	21	0	0
17	128	128	21	0	0
18	140	140	21	0	0
19	148	148	21	0	0
20-63	reserved				

1.8 Error protection

1.8.1 Overview of the tools

For error resilient audio object types, the error protection (EP) tool may be applied. The usage of this tool is signalled by the **epConfig** field. The input of the EP tool decoder consists of error protected access units. In case usage of the EP tool decoder is signalled by epConfig, the following restrictions apply:

- There exists one elementary stream per scalability layer, or just one elementary stream in case of non-scalable configurations.

- The output of the EP decoder is a set of several EP classes. The concatenation of EP classes at the output of the EP decoder is identical to epConfig = 0 data.

The definition of an EP class depends on **epConfig** and **directMapping**. For epConfig = 2, EP classes are not strictly defined. Their exact content is to be defined at application level, although the above mentioned restrictions have to be fulfilled. For epConfig = 3, the mapping between EP classes and instances of error sensitivity categories (ESCs) is normatively defined. In this case, the mapping is signalled by **directMapping**. In case directMapping = 1, each EP class maps exactly to one instance of an error sensitivity class. The EP decoder output then is identical to the case in which epConfig = 1. Figure 1.4 summarises the usage of EP classes, depending on the value of epConfig.

The error protection tool (EP tool) provides the unequal error protection (UEP) capability to the ISO/IEC 14496-3 codecs. The main features of the EP tool are as follows:

- providing a set of error correcting/detecting codes with wide and small-step scalability, in performance and in redundancy

- providing a generic and bandwidth-efficient error protection framework, which covers both fixed-length frame streams and variable-length frame streams

- providing a UEP configuration control with low overhead

The basic idea of UEP is to divide the frame into sub-frames according to the bit error sensitivities (these sub-frames are referred to be as classes in the following subclauses), and to protect these sub-frames with appropriate strength of FEC and/or CRC. If this would not be done, the decoded audio quality is determined by how the most

error sensitive part is corrupted, and thus the strongest FEC/CRC has to be applied to the whole frame, requiring much more redundancy.

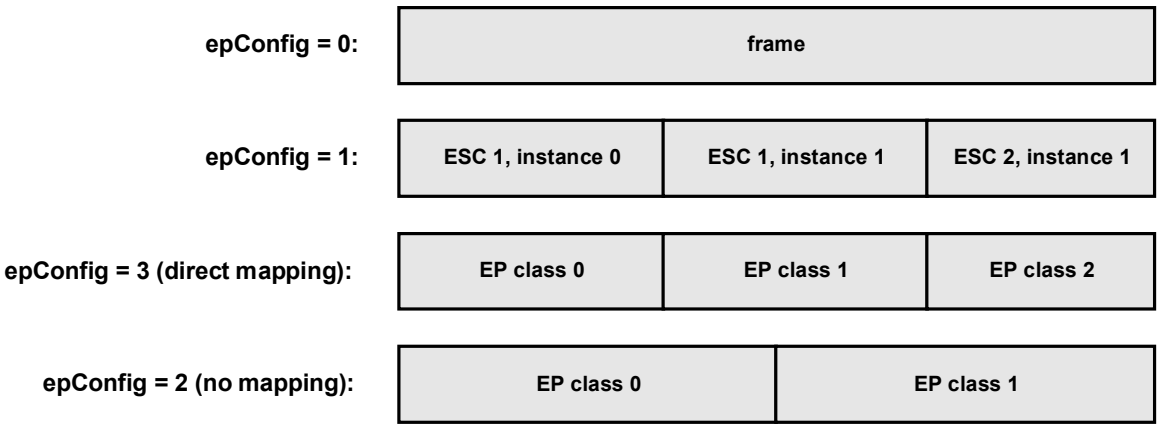


Figure 1.4 — EP classes for different epConfig values

In order to apply UEP to audio frames, the following information is required:

1) Number of classes

2) Number of bits each class contains

3) The CRC code to be applied for each class, which can be presented as a number of CRC bits

4) The FEC code to be applied for each class

This information is called as “frame configuration parameters” in the following sections. The same information is used to decode the UEP encoded frames; thus they have to be transmitted. To transmit them effectively, the frame structures of MPEG-4 audio algorithms have been taken into account for this EP tool.

The MPEG-4 audio frame structure can be categorized into three different approaches from the viewpoint of UEP application:

1) All the frame configurations are constant while the transmission (as CELP).

2) The frame configurations are restricted to be one of the several patterns (as Twin-VQ).

3) Most of the parameters are constant during the transmission, but some parameters can be different frame by frame (as AAC).

To utilize these characteristics, the EP tool uses two paths to transmit the frame configuration parameters. One is the out-of-band signaling, which is the same way as the transmission of codec configuration parameters. The parameters that are shared by the frames are transmitted through this path. In case there are several patterns of configuration, all these patterns are transmitted with indices. The other is the in-band transmission, which is made by defining the EP-frame structure with a header. Only the parameters that are not transmitted out-of-band are transmitted through this path. With this parameter transmission technique, the amount of in-band information, which is a part of the redundancy caused by the EP tool, is minimized.

With these parameters, each class is FEC/CRC encoded and decoded. To enhance the performance of this error protection, an interleaving technique is adopted. The objective of interleaving is to randomize burst errors within the frames, and this is not desirable for the class that is not protected. This is because there are other error resilience tools whose objective is to localize the effect of the errors, and randomization of errors with interleaving would have a harmful influence on such part of bitstream payload.

The outline of the EP encoder and EP decoder is figured out in Figure 1.5 and Figure 1.6.

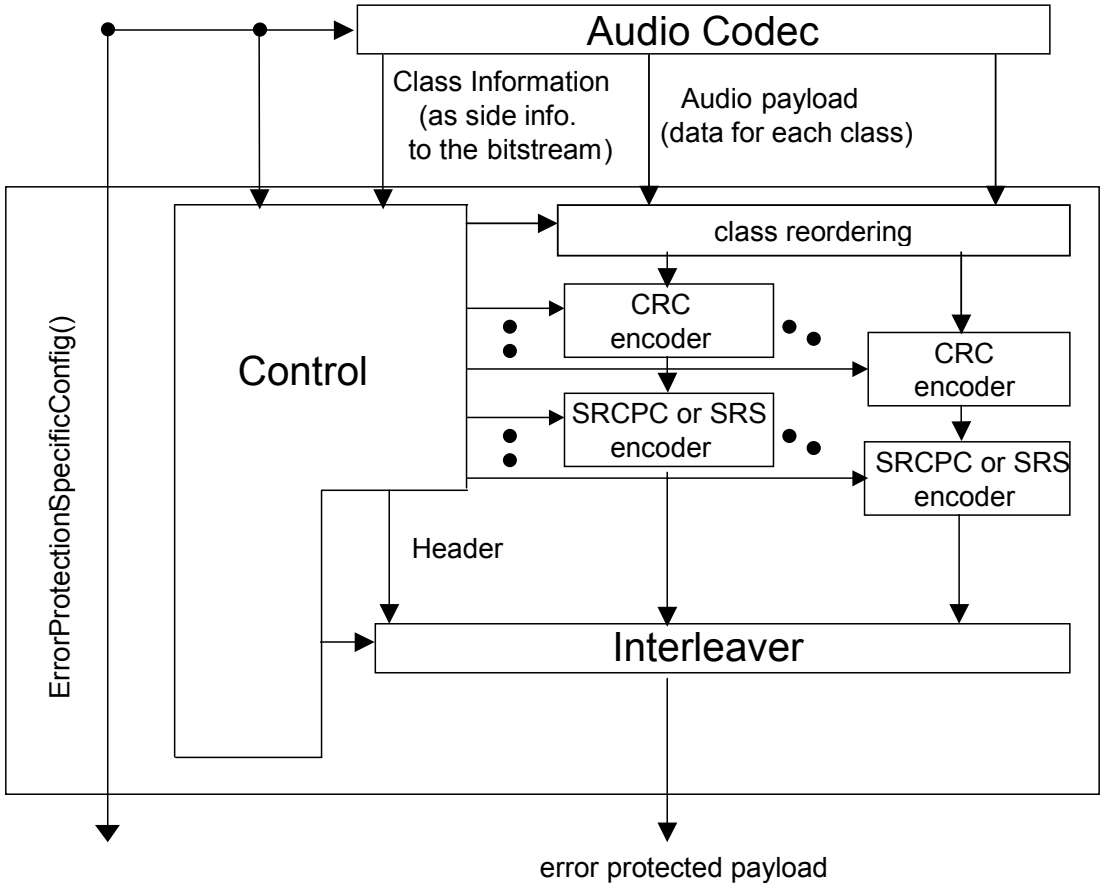


Figure 1.5 — Outline of EP encoder

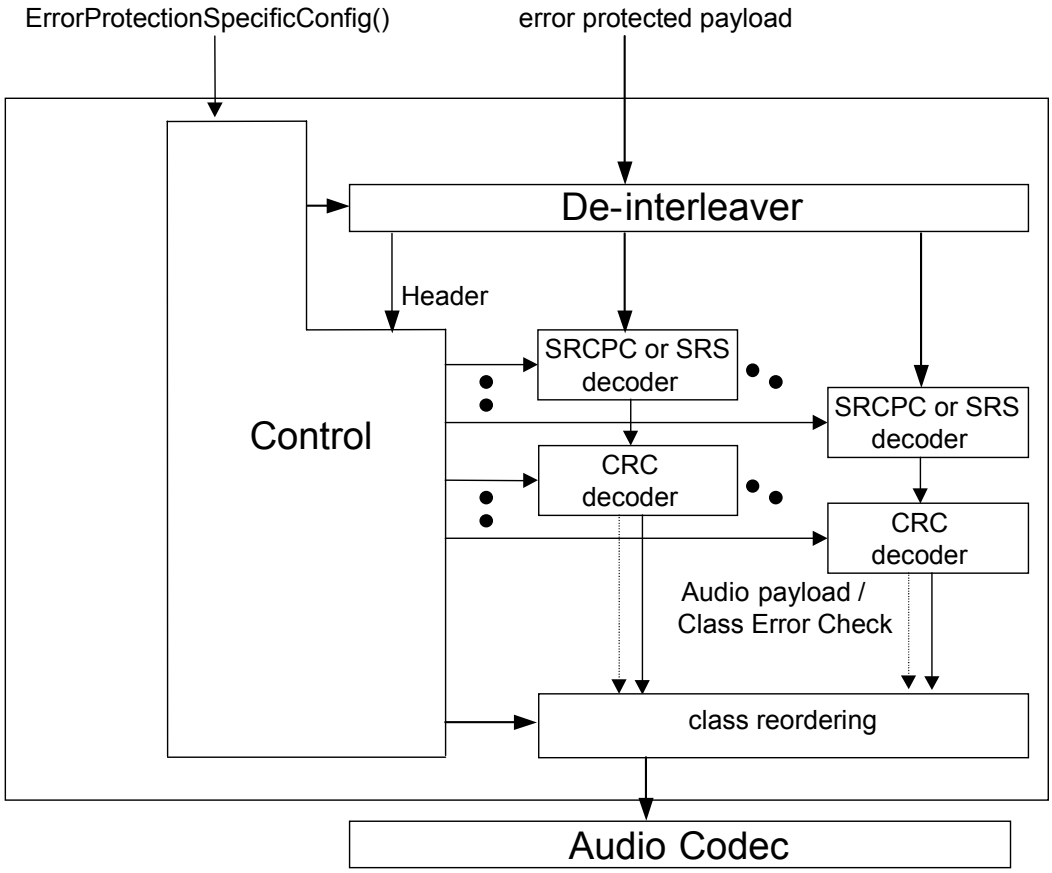


Figure 1.6 — Outline of EP decoder

1.8.2 Syntax

1.8.2.1 Error protection specific configuration

This part defines the syntax of the specific configuration for error protection.

Table 1.36 — Syntax of ErrorProtectionSpecificConfig()

Syntax	No. of bits	Mnemonic
ErrorProtectionSpecificConfig()		
{		
number_of_predefined_set;	**8**	**uimsbf**
interleave_type;	**2**	**uimsbf**
bit_stuffing;	**3**	**uimsbf**
number_of_concatenated_frame;	**3**	**uimsbf**
for (i = 0; i < number_of_predefined_set; i++) {		
number_of_class[i];	**6**	**uimsbf**
for (j = 0; j < number_of_class[i]; j++) {		
length_escape[i][j];	**1**	**uimsbf**
rate_escape[i][j];	**1**	**uimsbf**
crclen_escape[i][j];	**1**	**uimsbf**
if (number_of_concatenated_frame != 1) {		
concatenate_flag[i][j];	**1**	**uimsbf**

```
            }
            fec_type[i][j];                                  2          uimsbf
            if( fec_type[i][j] == 0) {
                termination_switch[i][j];                    1          uimsbf
            }
            if (interleave_type == 2) {
                interleave_switch[i][j];                     2          uimsbf
            }
            class_optional;                                  1          uimsbf
            if ( length_escape[i][j] == 1 ) {      /* ESC */
                number_of_bits_for_length[i][j];             4          uimsbf
            }
            else {
                class_length[i][j];                          16         uimsbf
            }
            if ( rate_escape[i][j] != 1 ) {    /* not ESC */
                if(fec_type[i][j]){
                    class_rate[i][j]                         7          uimsbf
                }else{
                    class_rate[i][j]                         5          uimsbf
                }
            }
            if ( crclen_escape[i][j] != 1 ) {       /* not ESC */
                class_crclen[i][j];                          5          uimsbf
            }
        }
        class_reordered_output;                              1          uimsbf
        if ( class_reordered_output == 1 ) {
            for ( j = 0; j < number_of_class[i]; j++ ) {
                class_output_order[i][j];                    6          uimsbf
            }
        }
    }
    header_protection;                                       1          uimsbf
    if ( header_protection == 1 ) {
        header_rate;                                         5          uimsbf
        header_crclen;                                       5          uimsbf
    }
}
```

1.8.2.2 Error protection bitstream payloads

This part defines the syntax of the error protected audio bitstream payload. This kind of syntax can be selected by setting epConfig=2 or epConfig=3. It is common for all audio object types. If MPEG-4 Systems is used, one ep_frame() is directly mapped to one access unit.

Table 1.37 — Syntax of ep_frame ()

Syntax	No. of bits	Mnemonic
ep_frame()		
{		
if (interleave_type == 0){		
ep_header();		
ep_encoded_classes();		
stuffing_bits;	**Nstuff**	**bslbf**
}		
if (interleave_type == 1){		
interleaved_frame_mode1;	**1 -**	**bslbf**
}		
if (interleave_type == 2){		
interleaved_frame_mode2;	**1 -**	**bslbf**
}		
}		

Nstuff: number of stuffing bits, identical to num_stuffing_bits, see subclause 1.8.3.1

Table 1.38 —Syntax of ep_header ()

Syntax	No. of bits	Mnemonic
ep_header()		
{		
choice_of_pred;	**N_{pred}**	**uimsbf**
choice_of_pred_parity;	**N_{pred_parity}**	**bslbf**
class_attrib();		
class_attrib_parity;	**N_{attrib_parity}**	**bslbf**
}		

Npred: ceil(log2 (number_of_predefined_set)).
Npred_parity: See subclause 1.8.4.3
Nattrib_parity: See subclause 1.8.4.3

Table 1.39 — Syntax of class_attrib ()

Syntax	No. of bits	Mnemonic
class_attrib()		
{		
for(j=0; j< number_of_class[choice_of_pred]; j++){		
if(class_reordered_output == 1){		
k = class_output_order[choice_of_pred][j];		
} else {		
k = j;		
}		
if (length_escape[choice_of_pred][k] == 1){		
class_bit_count[k];	**Nbitcount**	**uimsbf**
}		
if (rate_escape[choice_of_pred][k] == 1){		
class_code_rate[k];	**3**	**uimsbf**
}		
if (crclen_escape[choice_of_pred][k] == 1){		
class_crc_count[k];	**3**	**uimsbf**
}		
}		
if (bit_stuffing == 1){		
num_stuffing_bits;	**3**	**uimsbf**
}		
}		

Table 1.40 — Syntax of ep_encoded_classes()

Syntax	No. of bits	Mnemonic

ep_encoded_classes()
{
for(j=0; j< number_of_class[choice_of_pred]; j++){
if(class_reordered_output == 1){
k = class_output_order[choice_of_pred][j];
} else {
k = j;
}
ep_encoded_class[k];
}
}
		bslbf

1.8.3 General information

1.8.3.1 Definitions

ErrorProtectionSpecificConfig (): Error protection specific configuration that is out-of-band information.

number_of_predefined_set The number of pre-defined set.

interleave_type This variable defines the interleave type. (interleave_type == 0) means no interleaving, (interleave_type == 1) means intra-frame interleaving and (interleave_type == 2) enables interleaving fine tuning for each class. For details see subclause 1.8.4.8. (interleave_type==3) is reserved.

bit_stuffing Signals whether the bit stuffing to ensure the byte alignment is used with the in-band information or not:
1 indicates the bit stuffing is used.
0 indicates the bit stuffing is not used. This implies that the configuration provided with the out-of-band information ensure the EP-frame is byte-aligned.

number_of_concatenated_frame The number of concatenated source coder frames for the constitution of one error protected frame.

Table 1.41 — concatenated frames depending on number_of_concatenated_frame

Codeword	000	001	010	011	100	101	110	111
number of concatenated frame	reserved	1	2	3	4	5	6	7

number_of_class[i] The number of classes for i-th pre-defined set.

length_escape[i][j] If 0, the length of j-th class in i-th pre-defined set is fixed value. If 1, the length is variable. Note that in case “until the end”, this value should be 1, and the **number_of_bits_for_length[i][j]** value should be 0.

rate_escape[i][j] If 0, the SRCPC code rate of j-th class in i-th pre-defined set is fixed value. If 1, the code rate is signaled in-band.

crclen_escape[i][j] If 0, the CRC length of j-th class in i-th pre-defined set is fixed value. If 1, the CRC length is signaled in-band.

concatenate_flag[i][j] This parameter defines whether j-th class of i-th pre-defined set is concatenated or not. 0 indicates “not concatenated” and 1 indicates “concatenated”. (See subclause 1.8.4.4)

fec_type[i][j] This parameter defines whether SRCPC code (“0”) or RS code (“1” or “2”) are used to protect the j-th class of i-th pre-defined set. Note that the class length which is signaled to be protected by RS code shall be byte aligned, in either case that the length is signaled in the out-of-band information or that the length is signaled as in-band information. If this field is set to “2”, it indicates that this class is RS encoded in conjunction with next class as one RS code. Note that more than two succeeding classes have the value “2” for this field, it means these classes are concatenated and RS encoded as one RS code. If this field is “1”, it indicates that this class is not concatenated with next class. This means this class is the last class to be concatenated before RS encoding, or this class is RS encoded independently.

termination_switch[i][j] This parameter defines whether j-th class of i-th pre-defined set is terminated or not when it is SRCPC encoded. See subclause 1.8.4.6.2.

interleave_switch[i][j] This parameter defines how to interleave j-th class of i-th pre-defined set.
0 – not interleaved
1 – interleaved without intraclass-interleaving: the interleaving width is same as the number of bits within the current class if (fec_type == 0), but same as the number of bytes within the current class if (fec_type==1 || fec_type == 2)
2 – interleaved with intraclass-interleaving: interleaving width is 28 if (fec_type == 0); but this value is reserved if (fec_type == 1 || fec_type == 2)
3 – concatenated
(see subclause 1.8.4.8.2.2)

class_optional This flag signals, whether the class is mandatory (class_optional == 0) or optional (class_optional == 1). This flag can be used to reduce the redundancy within ErrorProtectionSpecificConfig. Usually it would be necessary to define 2^N predefinition sets, where N equals the number of optional classes (see subclause 1.8.4.2).

number_of_bits_for_length[i][j] This field exists only when the **length_escape[i][j]** is 1. This value shows the number of bits for the class length in-band signaling. This value should be set considering possible maximum length of the class. The value 0 indicates the “until the end” functionality (see subclause 1.8.4.1).

class_length[i][j] This field exists only when the **length_escape[i][j]** is 0. This value shows the length of the j-th class in i-th pre-defined set, which is the fixed value while the transmission.

class_rate[i][j] This field exists only when the **rate_escape[i][j]** is 0. In case **fec_type[i][j]** is 0, this value shows the SRCPC code rate of the j-th class in i-th pre-defined set, which is the fixed value while the transmission. The value from 0 to 24 corresponds to the code rate from 8/8 to 8/32, respectively. In case **fec_type[i][j]** is 1 or 2, this value shows the number of erroneous bytes which can be corrected by RS code (see subclause 1.8.4.7). All the classes which is signaled to be concatenated with **fec_type[i][j]** shall have the same value of **class_rate[i][j]**.

class_crclen[i][j] This field exists only when the **crclen_escape[i][j]** is 0. This value shows the CRC length of the j-th class in i-th pre-defined set, which is the fixed value while the transmission. The value should be 0 – 18, which represents CRC length 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 24 or 32. (See subclause1.8.4.5)

class_reordered_output If this value is “1”, the classes output from ep decoder is re-ordered. If “0”, no such processing is made. See subclause 1.8.4.9.

class_output_order[i][j] This field only exists when class_reordered_output is set to “1”, to signal the order of the class after re-ordering. The j-th class of i-th pre-defined set is output as (**class_output_order[i][j]**)-th class from ep decoder. See subclause 1.8.4.9.

header_protection This value indicates the header error protection mode. 0 indicates the use of basic set of FEC, and 1 indicates the use of extended header error protection, as defined in subclause 1.8.4.3. The extended header error protection is applied only if the length of the header exceeds 16 bits.

header_rate, header_crclen These values have the same semantics with **class_rate[i][j]** and **class_crclen[i][j]** respectively, while these error protection is utilized for the protection of header part.

ep_frame() error protected frame.

ep_header() EP frame header information.

ep_encoded_classes() The EP encoded audio information.

interleaved_frame_mode1 The information bits after interleaving with interleaving mode 1. See subclause 1.8.4.1 and subclause 1.8.4.8.

interleaved_frame_mode2 The information bits after interleaving with interleaving mode 2. See subclause 1.8.4.1 and subclause 1.8.4.8.

stuffing_bits The stuffing bits for the EP frame octet alignment. The number of bits Nstuff is signaled in class_attrib(), and should be in the range of 0...7.

choice_of_pred The choice of pre-defined set. See subclause 1.8.4.2.

choice_of_pred_parity The parity bits for **choice_of_pred**. See subclause 1.8.4.2.

class_attrib_parity The parity bits for class_attrib(). See subclause 1.8.4.2.

class_attrib() Attribution information for each class

class_bit_count[j] The number of information bits included in the class. This field only exists in case the length_escape in out-of-band information is 1 (escape). The number of bits of this parameter Nbitcount is also signaled in the out-of-band information.

class_code_rate[j] The coding rate for the audio data belonging to the class, as defined in the table below. This field only exists in case the rate_escape in out-of-band information is 1 (escape).

Table 1.42 — The coding rate for the audio data belonging to the class

Codeword	000	001	010	011	100	101	110	111
Puncture Rate	8/8	8/11	8/12	8/14	8/16	8/20	8/24	8/32
Puncture Pattern	FF, 00 00, 00	FF, A8 00, 00	FF, AA 00, 00	FF, EE 00, 00	FF, FF 00, 00	FF, FF AA, 00	FF, FF FF, 00	FF, FF FF, FF

class_crc_count[j] The number of CRC bits for the audio data belonging to the class, as defined in the table below. This field only exists in case the crclen_escape in out-of-band information is 1 (escape).

Table 1.43 — The number of CRC bits for the audio data belonging to the class

Codeword	000	001	010	011	100	101	110	111
CRC bits	0	6	8	10	12	14	16	32

num_stuffing_bits the number of stuffing bits for the EP frame octet alignment. This field only exists in case the **bit_stuffing** in out-of-band information is 1.

ep_encoded_class[j] CRC/SRCPC encoded audio data of j-th class.Note that if **class_bit_count[j]** == 0, audio data of j-th class is not encoded by CRC/SRCPC/SRS.

1.8.4 Tool description

1.8.4.1 Out-of-band information

The content of out-of band information is represented by means of ErrorProtectionSpecificConfig(). Some configuration examples are provided in Annex 1.B.

The length of the last class processed by the EP decoder (prior to any subsequent re-ordering as described in subclause 1.8.4.9) does not have to be transmitted explicitly, but its length might be signaled as “until the end”. In MPEG-4 Systems, the systems layer guarantees the audio frame boundary by mapping one audio frame to one access unit. Therefore, the length of the “until the end” class can be calculated from the length of other classes and the total EP-encoded audio frame length.

The flag class_optional might be used to reduce the redundancy within ErrorProtectionSpecificConfig(). However, the EP tool still works with the same number of pre-defined sets. If there are N classes with (class_optional == 1), this pre-defined set is extended to 2^N pre-defined sets. Unwrapping of the predefinition sets is described within the following subclause.

1.8.4.2 Derivation of pre-defined sets

This subclause describes the post processing, whose input is ErrorProtectionSpecificConfig() with “class_optional” switch and whose output are pre-defined sets used for the ep_frame() parameters.

General procedure:

- Each pre-defined set expands $2^{NCO[i]}$ pre-defined sets, where NCO[i] is the number of classes with (class_optional == 1) in the i-th original pre-defined set. Hereafter, any class with (class_optional == 1) is referred to as optClass.

- These expanded pre-defined sets start from “all the optClasses exist” to “all the optClasses do not exist”.

Algorithm:

```
transPred = 0;
for ( i = 0; i < nPred; i++ ) {          /* for all predefinition sets */
   for ( j = 0; j < 2^NCO[i]; j++ ) {       /* unwraping */
      for ( k = 0; k < NCO[i]; k++ ) {      /* for all optional classes */
         if ( j & ( 0x01 << k ) ) {
            optClassExists[k] = 0;
         }
         else {
            optClassExists[k] = 1;
         }
      }
      DefineTransPred(transPred, i, optClassExists);
      transPred ++;
   }
}
```

where,

optClassExists[k] signals whether k-th optClass of the pre-defined set exists (1) or not (0) in the defining new pre-defined set.

DefineTransPred (transPred, i, optClassExists) defines transPred-th new pre-defined set used for the transmission. This new pre-defined set is a copy of the i-th original pre-defined set, except it does not have optClasses whose optClassExists equals to 0.

Example

ErrorProtectionSpecificConfig() defines pre-defined sets as follows:

Table 1.44 — The example of pre-defined set

Pred #0		Pred #1	
Class A	class_optional = 1	Class F	class_optional = 1
Class B	class_optional = 0	Class G	class_optional = 0
Class C	class_optional = 1		
Class D	class_optional = 0		
Class E	class_optional = 1		

After the pre-processing described above, the pre-defined sets used for ep_frame() becomes as follows:

Table 1.45 — The example of pre-defined sets after the pre-processing

Pred #0	Pred #1	Pred #2	Pred #3	Pred #4	Pred #5	Pred #6	Pred #7	Pred #8	Pred #9
Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B	Class F	Class G
Class B	Class C	Class B	Class D	Class B	Class C	Class B	Class D	Class G	
Class C	Class D	Class D	Class E	Class C	Class D	Class D			
Class D	Class E	Class E		Class D					
Class E									

1.8.4.3 In-band information

The EP frame information, which is not included in the out-of-band information, is the in-band information. The parameters belonging to this information are transmitted as an EP frame header. The parameters are:

- The choice of pre-defined set
- The number of stuffing bits for byte alignment
- The class information which is not included in the out-of-band information

The EP decoder cannot decode the audio frame information without these parameters, and thus they have to be error protected stronger than or equal to the other parts. On this error protection, the choice of pre-defined set has to be treated differently from the other parts. This is because the length of the class information can be changed according to which pre-defined set is chosen. For this reason, this parameter is FEC encoded independently from the other parts. At decoder side, the choice of pre-defined set is decoded first, and then the length of the remaining header part is calculated with this information, and decodes that.

The FEC applied for these parts are as follows:

Basic set of FEC codes:

Table 1.46 — Basic set of FEC codes for in-band information

number of bits to be protected	FEC code	total number of bits	Length of codeword
1-2	majority (repeat 3 times)	3-6	3
3-4	BCH(7,4)	6-7	6-7
5-7	BCH(15,7)	13-15	13-15
8-12	Golay(23,12)	19-23	19-23
13-16	BCH(31,16)	28-31	28-31
17-	RCPC 8/16 + 4-bit CRC	50 -	-

Notes:

- total number of bits: number of bits after FEC encoding
- interleaving width: width of the interleaving matrix, see also subclause 1.8.4.8
- Npred_parity (or Nattrib_parity) = total number of bits – number of bits to be protected
- SRCPC is terminated
- number of bits to be protected is Npred (or the total number of bits for class_attrib())

Extended FEC:

If a header length exceeds 16 bits, this header is protected using a CRC and a terminated SRCPC. The SRCPC code rate and the number of CRC bits are signaled. The encoding and decoding method for this is the same as described below within the CRC/SRCPC description.

The generator polynomials for each FEC are as follows:

BCH(7,4): x^3+x+1

BCH(15,7): $x^8+x^7+x^6+x^4+1$

Golay(23,12): $x^{11}+x^9+x^7+x^6+x^5+x+1$

BCH(31,16): $x^{15}+x^{11}+x^{10}+x^9+x^8+x^7+x^5+x^3+x^2+x+1$

With these polynomials, the FEC (*n, k*) for *l*-bit information encoding is made as follows:

Calculate the polynomial R(x) that satisfies

$M(x)\, x^{n-l} = Q(x)G(x) + R(x)$

M(x): Information bits. Highest order corresponds to the first bit to be transmitted

G(x): The generation polynomial from the above definition

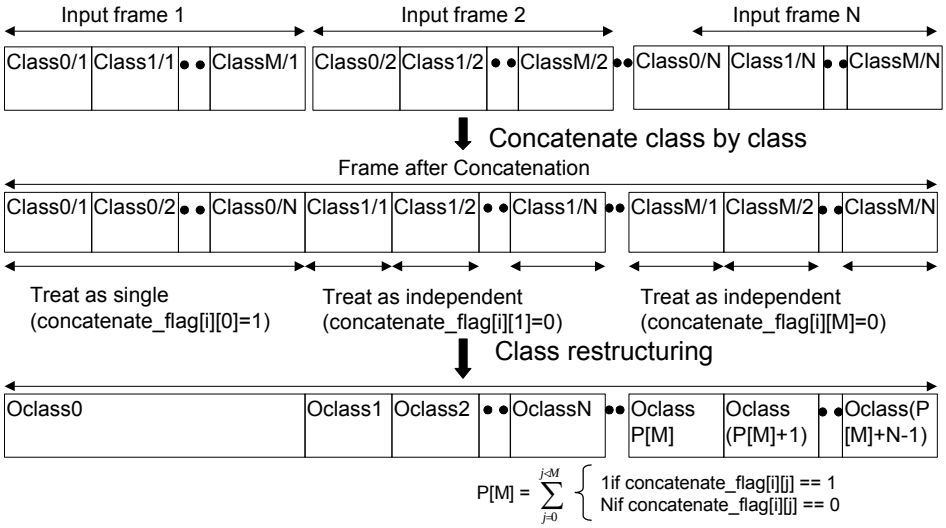
This polynomial R(x) represents parity to choice_of_pred or class_attrib(), and set to choice_of_pred_parity or class_attrib_parity respectively. The highest order corresponds to the first bit. The decoder can perform error correction using these parity bits , while it is optional operation.

1.8.4.4 Concatenation functionality

EP tool has a functionality to concatenate several source coder frames to build up a new frame for the EP tool. In this concatenation, the groups of bits belonging to the same class in the different source coder frames are concatenated in adjacent, class by class basis. The concatenated groups belonging to the same class is either treated as a single new one class or independent class in the same manner as before the concatenation.

The number of frames to be concatenated is signaled as number_of_concatenated_frame in ErrorProtectionSpecificConfig(), and the choice whether the concatenated groups belonging to the same class is treated as single new one class or independent class is signaled by concatenate_flag[i][j] (1 indicate “single new one class”, and 0 indicates “independent class”). This process is illustrated in Figure 1.7.

The same pre-defined set shall be used for all concatenated frames. No escape mechanism shall be used for any class parameter.



*Oclass: Output class to be proceed for CRC/FEC encoding

Figure 1.7 — Concatenation procedure

1.8.4.5 CRC

The CRC provides error detection capability. The information bits of each class is CRC encoded as a first process. In this tool, the following set of the CRC is defined:

1-bit CRC ***CRC1***: $x+1$

2-bit CRC ***CRC2***: x^2+x+1

3-bit CRC ***CRC3***: x^3+x+1

4-bit CRC ***CRC4***: $x^4+x^3+x^2+1$

5-bit CRC ***CRC5***: $x^5+x^4+x^2+1$

6-bit CRC ***CRC6*** : $x^6+x^5+x^3+x^2+x+1$

7-bit CRC ***CRC7*** : $x^7+x^6+x^2+1$

8-bit CRC ***CRC8*** : x^8+x^2+x+1

9-bit CRC ***CRC9*** : $x^9+x^8+x^5+x^2+x+1$

10-bit CRC ***CRC10*** : $x^{10}+x^9+x^5+x^4+x+1$

11-bit CRC ***CRC11*** : $x^{11}+x^{10}+x^4+x^3+x+1$

12-bit CRC ***CRC12*** : $x^{12}+x^{11}+x^3+x^2+x+1$

13-bit CRC ***CRC13*** : $x^{13}+x^{12}+x^7+x^6+x^5+x^4+x^2+1$

14-bit CRC ***CRC14*** : $x^{14}+x^{13}+x^5+x^3+x^2+1$

15-bit CRC ***CRC15*** : $x^{15}+x^{14}+x^{11}+x^{10}+x^7+x^6+x^2+1$

16-bit CRC ***CRC16*** : $x^{16}+x^{12}+x^5+1$

24-bit CRC ***CRC24*** : $x^{24}+x^{23}+x^6+x^5+x+1$

32-bit CRC ***CRC32*** : $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

With these polynomials, the CRC encoding is made as follows:

Calculate the polynomial R(x) that satisfies

$M(x)x^k = Q(x)G(x) + R(x)$

M(x): Information bits. Highest order corresponds to the first bit to be transmitted

G(x): The generation polynomial from the above definition

k: The number of CRC bits.

With this polynomial *R(x),* the CRC encoded bits *W(x)* is represented as:

$W(x) = M(x)x^k + R(x)$

Note that the value k should be chosen so that the number of CRC encoded bits does not exceed 2k-1.

The CRC bits are written in a reversed manner, i. e. each bit is inverted. Using these CRC bits, the decoder can perform error detection. When an error is detected through CRC, error concealment may be applied to reduce the quality degradation caused by the error. The error concealment method depends on MPEG-4 audio algorithms, an example is given in subclause 1.B.3.

1.8.4.6 Systematic rate-compatible punctured convolutional (SRCPC) codes

Following to the CRC encoding, FEC encoding is made with the SRCPC codes. This subclause describes the SRCPC encoding process.

The channel encoder is based on a systematic recursive convolutional (SRC) encoder with rate R=1/4. The CRC encoded classes are concatenated and input into this encoder. Then, with the puncturing procedure described in the subclause later, we obtain a Rate Compatible Punctured Convolutional (RCPC) code whose code rate varies for each class according to the error sensitivity.

1.8.4.6.1 SRC code generation

The SRC code is generated from a rational generator matrix by using a feedback loop. A shift register realization of the encoder is shown in Figure 1.8.

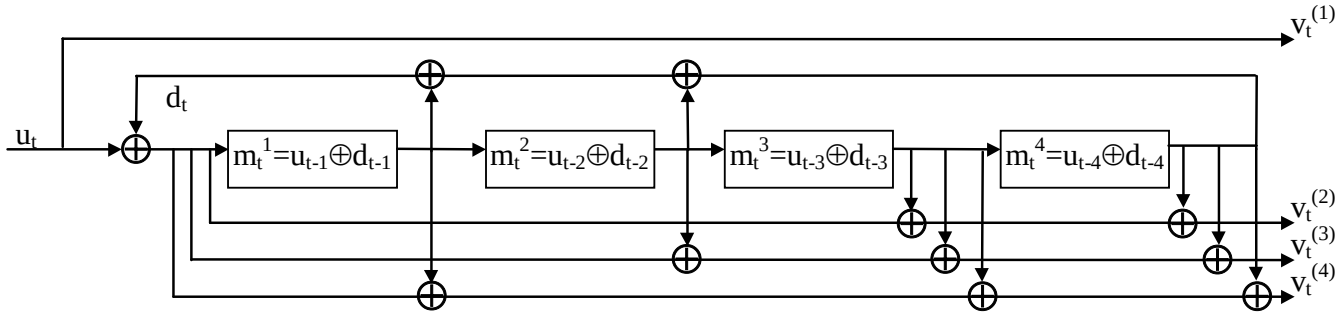


Figure 1.8 — Shift register realization for systematic recursive convolutional encoder

To obtain the output vectors v_t at each time instant t, one has to know the content of the shift registers m_t^1, m_t^2, m_t^3, m_t^4 (corresponds to the state) and the input bit u_t at time t.

We obtain the output $v_t^{(2)}$, $v_t^{(3)}$ and $v_t^{(4)}$

$$v_t^{(2)} = m_t^4 \oplus m_t^3 \oplus (u_t \oplus d_t)$$

$$v_t^{(3)} = m_t^4 \oplus m_t^3 \oplus m_t^2 \oplus (u_t \oplus d_t)$$

$$v_t^{(4)} = m_t^4 \oplus m_t^3 \oplus m_t^1 \oplus (u_t \oplus d_t)$$

with

$$d_t = m_t^4 \oplus m_t^2 \oplus m_t^1,\ m_t^4 = u_{t-4} \oplus d_{t-4},\ m_t^3 = u_{t-3} \oplus d_{t-3},\ m_t^2 = u_{t-2} \oplus d_{t-2},\ m_t^1 = u_{t-1} \oplus d_{t-1}$$

Finally we obtain for the output vector $\underline{\boldsymbol{v}}_t = (v_t^{(1)}, v_t^{(2)}, v_t^{(3)}, v_t^{(4)})$ at time t depending on the input bit u_t and the current state $\underline{\boldsymbol{m}}_t = (m_t^1, m_t^2, m_t^3, m_t^4)$:

$$\begin{aligned}
V_t^{(1)} &= u_t \\
V_t^{(2)} &= m_t^4 \oplus m_t^3 \oplus (u_t \oplus d_t) = m_t^3 \oplus m_t^2 \oplus m_t^1 \oplus u_t \\
V_t^{(3)} &= m_t^4 \oplus m_t^3 \oplus m_t^2 \oplus (u_t \oplus d_t) = m_t^3 \oplus m_t^1 \oplus u_t \\
V_t^{(4)} &= m_t^4 \oplus m_t^3 \oplus m_t^1 \oplus (u_t \oplus d_t) = m_t^3 \oplus m_t^2 \oplus u_t
\end{aligned}$$

with $\underline{\mathbf{m}}_1 = (m_1^1, m_1^2, m_1^3, m_1^4) = (0, 0, 0, 0) = \underline{\mathbf{0}}$

The initial state is always $\underline{\mathbf{0}}$, i.e. each memory cell contains a 0 before the input of the first information bit u_t.

1.8.4.6.2 Termination of SRC code

In case the SRC coded class is indicated as terminated with termination_switch[i] in ErrorProtectionSpecificConfig(), or SRC code is used for the protection of in-band information, the SRC encoder shall add the tail bits at the end of this class, and start the succeeding SRC encoding with initial state (all the encoder shift register shall reset to be 0).

The tail bits following the information sequence $\boldsymbol{u}$ for returning to state $\underline{\boldsymbol{m}}_n = \underline{\boldsymbol{0}}$ (termination) depends on the last state $\underline{\boldsymbol{m}}_{n-3}$ (state after the input of the last information bit u_{n-4}.). The termination sequence for each state described by $\underline{\boldsymbol{m}}_{n-3}$ is given in Table 1.47. The receiver may use these tail bits (TB) for additional error detection.

The appendix $(u_{n-3}, u_{n-2}, u_{n-1}\ u_n)$ to the information sequence can be calculated with the following condition:

$$\text{for all } t \text{ with } n-3 \le t \le n: u_t \oplus d_t = 0$$

Hence we obtain for the tail bit vector $\underline{\boldsymbol{u}}'=(u_{n-3}, u_{n-2}, u_{n-1}, u_n)$ depending on the state $\underline{\boldsymbol{m}}_{n-3} =(m_{n-3}^{1}, m_{n-3}^{2}, m_{n-3}^{3}, m_{n-3}^{4})$

$$u_{n-3} = d_{n-3} = m_{n-3}^{4} \oplus m_{n-3}^{2} \oplus m_{n-3}^{1}$$
$$u_{n-2} = d_{n-2} = m_{n-2}^{4} \oplus m_{n-2}^{2} \oplus m_{n-2}^{1} = m_{n-3}^{3} \oplus m_{n-3}^{1} \oplus 0 = m_{n-3}^{3} \oplus m_{n-3}^{1}$$
$$u_{n-1} = d_{n-1} = m_{n-1}^{4} \oplus m_{n-1}^{3} \oplus m_{n-1}^{2} = m_{n-3}^{2} \oplus 0 \oplus 0 = m_{n-3}^{2}$$
$$u_n = d_n = m_{n-3}^{1} \oplus 0 \oplus 0 = m_{n-3}^{1}$$

Table 1.47 — Tail bits for systematic recursive convolutional code

state $\underline{m}_{n-3}$	m_{n-3}^{4}	m_{n-3}^{3}	m_{n-3}^{2}	m_{n-3}^{1}	u_{n-3}	u_{n-2}	u_{n-1}	u_n
0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	0	1
2	0	0	1	0	1	0	1	0
3	0	0	1	1	0	1	1	1
4	0	1	0	0	0	1	0	0
5	0	1	0	1	1	0	0	1
6	0	1	1	0	1	1	1	0
7	0	1	1	1	0	0	1	1
8	1	0	0	0	1	0	0	0
9	1	0	0	1	0	1	0	1
10	1	0	1	0	0	0	1	0
11	1	0	1	1	1	1	1	1
12	1	1	0	0	1	1	0	0
13	1	1	0	1	0	0	0	1
14	1	1	1	0	0	1	1	0
15	1	1	1	1	1	0	1	1

1.8.4.6.3 Puncturing of SRC for SRCPC code

Puncturing of the output of the SRC encoder allows different rates for transmission. The puncturing tables are listed in Table 1.48.

Table 1.48 — Puncturing tables (all values in hexadecimal representation)

Rate r	8/8	8/9	8/10	8/11	8/12	8/13	8/14	8/15	8/16	8/17	8/18	8/19	8/20
$P_r(0)$	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
$P_r(1)$	00	80	88	A8	AA	EA	EE	FE	FF	FF	FF	FF	FF
$P_r(2)$	00	00	00	00	00	00	00	00	00	80	88	A8	AA
$P_r(3)$	00	00	00	00	00	00	00	00	00	00	00	00	00

Rate r	8/21	8/22	8/23	8/24	8/25	8/26	8/27	8/28	8/29	8/30	8/31	8/32
$P_r(0)$	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
$P_r(1)$	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
$P_r(2)$	EA	EE	FE	FF	FF	FF	FF	FF	FF	FF	FF	FF
$P_r(3)$	00	00	00	00	80	88	A8	AA	EA	EE	FE	FF

The puncturing pattern, which is applied with the period of 8, depends on the class_rate (see Table 1.48). Each bit of Pr(i) indicates whether the corresponding vt(i) from the SRC encoder is punctured (i.e. not considered). Each bit of Pr(i) is used from MSB to LSB, and 0/1 indicates punctured/not-punctured respectively. The puncturing pattern

changes class-wise, but only at points where a period of 8 is completed. After the decision which bits from vt(i) are considered, they are output in the order from vt(0) to vt(3).

1.8.4.6.4 Decoding process of SRCPC code

At the decoder, the error correction should be performed using this SRCPC code, while it is the optional operation and the decoder may extract the original information by just ignoring parity bits.

Decoding of SRCPC can be achieved using Viterbi algorithm for the punctured convolutional coding.

1.8.4.7 Shortened Reed-Solomon codes

Shortened RS codes SRS(255-*l*, 255-2*k*-*l*) defined over GF(2^8) can be used to protect one single class or several concatenated classes. Concatenated classes are subsequently treated as one single class. Here, *k* is the number of correctable errors in one SRS codeword. The value of *l* reflects the shortening.

Before the SRS encoding, the EP class is sub-divided into parts such that their lengths are less than or equal to 255-2*k*. The lengths of the parts are calculated as follows:

l_i = 255-2*k*, for *i*<*N*

l_i = *L* mod (255-2*k*), for *i* = *N*

N = *ceil* (*L* / (255-2*k*))

L: The length of the EP class in octets

N: The number of parts

l_i: The length of the i-th part (0<i<N+1)

If the length of the N-th part l_N is smaller than 255-2*k* bytes, as many bits with the value 0 are added as needed to reach the length of 255-2k bytes before SRS en- or decoding, and again removed afterwards.

At decoder side, if SRS decoding is performed, the same number of '0's have to be added before the SRS decoding procedure, and removed again after SRS decoding.

The SRS code defined in the Galois Field GF(2^8) is generated from a generator polynomial $g(x) = (x-\alpha)(x-\alpha^2)\cdots(x-\alpha^{2k})$, where α denotes a root of the primitive polynomial $m(x)=x^8+x^4+x^3+x^2+1$. The binary representative of α^j is shown in Table 1.49, where the MSB of the octet is transmitted first.

Table 1.49 — Binary representation for α^i ($0 \le i \le 254$) over GF(2^8)

a^l	binary rep.	a^l	binary rep.	a^l	binary rep.	a^l	binary rep.
0	00000000	a^{63}	10100001	a^{127}	11001100	a^{191}	01000001
a^0	00000001	a^{64}	01011111	a^{128}	10000101	a^{192}	10000010
a^1	00000010	a^{65}	10111110	a^{129}	00010111	a^{193}	00011001
a^2	00000100	a^{66}	01100001	a^{130}	00101110	a^{194}	00110010
a^3	00001000	a^{67}	11000010	a^{131}	01011100	a^{195}	01100100
a^4	00010000	a^{68}	10011001	a^{132}	10111000	a^{196}	11001000
a^5	00100000	a^{69}	00101111	a^{133}	01101101	a^{197}	10001101
a^6	01000000	a^{70}	01011110	a^{134}	11011010	a^{198}	00000111
a^7	10000000	a^{71}	10111100	a^{135}	10101001	a^{199}	00001110
a^8	00011101	a^{72}	01100101	a^{136}	01001111	a^{200}	00011100
a^9	00111010	a^{73}	11001010	a^{137}	10011110	a^{201}	00111000
a^{10}	01110100	a^{74}	10001001	a^{138}	00100001	a^{202}	01110000
a^{11}	11101000	a^{75}	00001111	a^{139}	01000010	a^{203}	11100000
a^{12}	11001101	a^{76}	00011110	a^{140}	10000100	a^{204}	11011101
a^{13}	10000111	a^{77}	00111100	a^{141}	00010101	a^{205}	10100111
a^{14}	00010011	a^{78}	01111000	a^{142}	00101010	a^{206}	01010011
a^{15}	00100110	a^{79}	11110000	a^{143}	01010100	a^{207}	10100110
a^{16}	01001100	a^{80}	11111101	a^{144}	10101000	a^{208}	01010001
a^{17}	10011000	a^{81}	11100111	a^{145}	01001101	a^{209}	10100010
a^{18}	00101101	a^{82}	11010011	a^{146}	10011010	a^{210}	01011001
a^{19}	01011010	a^{83}	10111011	a^{147}	00101001	a^{211}	10110010
a^{20}	10110100	a^{84}	01101011	a^{148}	01010010	a^{212}	01111001
a^{21}	01110101	a^{85}	11010110	a^{149}	10100100	a^{213}	11110010
a^{22}	11101010	a^{86}	10110001	a^{150}	01010101	a^{214}	11111001
a^{23}	11001001	a^{87}	01111111	a^{151}	10101010	a^{215}	11101111
a^{24}	10001111	a^{88}	11111110	a^{152}	01001001	a^{216}	11000011
a^{25}	00000011	a^{89}	11100001	a^{153}	10010010	a^{217}	10011011
a^{26}	00000110	a^{90}	11011111	a^{154}	00111001	a^{218}	00101011
a^{27}	00001100	a^{91}	10100011	a^{155}	01110010	a^{219}	01010110
a^{28}	00011000	a^{92}	01011011	a^{156}	11100100	a^{220}	10101100
a^{29}	00110000	a^{93}	10110110	a^{157}	11010101	a^{221}	01000101
a^{30}	01100000	a^{94}	01110001	a^{158}	10110111	a^{222}	10001010
a^{31}	11000000	a^{95}	11100010	a^{159}	01110011	a^{223}	00001001
a^{32}	10011101	a^{96}	11011001	a^{160}	11100110	a^{224}	00010010
a^{33}	00100111	a^{97}	10101111	a^{161}	11010001	a^{225}	00100100
a^{34}	01001110	a^{98}	01000011	a^{162}	10111111	a^{226}	01001000
a^{35}	10011100	a^{99}	10000110	a^{163}	01100011	a^{227}	10010000
a^{36}	00100101	a^{100}	00010001	a^{164}	11000110	a^{228}	00111101
a^{37}	01001010	a^{101}	00100010	a^{165}	10010001	a^{229}	01111010
a^{38}	10010100	a^{102}	01000100	a^{166}	00111111	a^{230}	11110100
a^{39}	00110101	a^{103}	10001000	a^{167}	01111110	a^{231}	11110101
a^{40}	01101010	a^{104}	00001101	a^{168}	11111100	a^{232}	11110111
a^{41}	11010100	a^{105}	00011010	a^{169}	11100101	a^{233}	11110011
a^{42}	10110101	a^{106}	00110100	a^{170}	11010111	a^{234}	11111011
a^{43}	01110111	a^{107}	01101000	a^{171}	10110011	a^{235}	11101011
a^{44}	11101110	a^{108}	11010000	a^{172}	01111011	a^{236}	11001011
a^{45}	11000001	a^{109}	10111101	a^{173}	11110110	a^{237}	10001011
a^{46}	10011111	a^{110}	01100111	a^{174}	11110001	a^{238}	00001011
a^{47}	00100011	a^{111}	11001110	a^{175}	11111111	a^{239}	00010110
a^{48}	01000110	a^{112}	10000001	a^{176}	11100011	a^{240}	00101100
a^{49}	10001100	a^{113}	00011111	a^{177}	11011011	a^{241}	01011000
a^{50}	00000101	a^{114}	00111110	a^{178}	10101011	a^{242}	10110000
a^{51}	00001010	a^{115}	01111100	a^{179}	01001011	a^{243}	01111101

a^{52}	00010100	a^{116}	11111000	a^{180}	10010110	a^{244}	11111010
a^{53}	00101000	a^{117}	11101101	a^{181}	00110001	a^{245}	11101001
a^{54}	01010000	a^{118}	11000111	a^{182}	01100010	a^{246}	11001111
a^{55}	10100000	a^{119}	10010011	a^{183}	11000100	a^{247}	10000011
a^{56}	01011101	a^{120}	00111011	a^{184}	10010101	a^{248}	00011011
a^{57}	10111010	a^{121}	01110110	a^{185}	00110111	a^{249}	00110110
a^{58}	01101001	a^{122}	11101100	a^{186}	01101110	a^{250}	01101100
a^{59}	11010010	a^{123}	11000101	a^{187}	11011100	a^{251}	11011000
a^{60}	10111001	a^{124}	10010111	a^{188}	10100101	a^{252}	10101101
a^{61}	01101111	a^{125}	00110011	a^{189}	01010111	a^{253}	01000111
a^{62}	11011110	a^{126}	01100110	a^{190}	10101110	a^{254}	10001110

For each of the parts, the SRS parity digits with a total length of 2*k* octets are calculated using $g(x)$ as follows:

$p(x) = x^{2k} \cdot u(x) \bmod g(x)$

$u(x)$: polynomial representative of a part. Lowest order corresponds to the first octet.

$p(x)$: polynomial representative of the parity digits. Lowest order corresponds to the first octet.

For storage and transmission, the parity digits are appended at the end of the EP class. This process is illustrated in Figure 1.9.

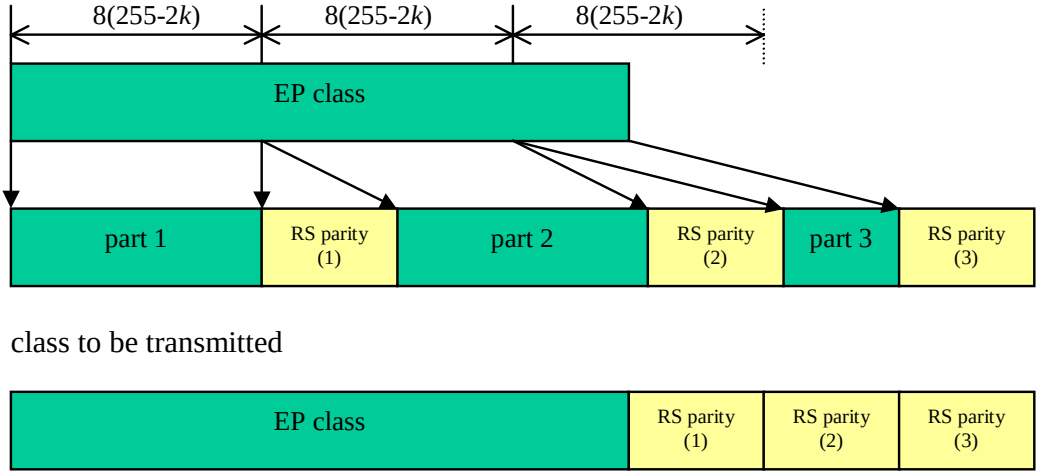


Figure 1.9 — RS encoding of EP frame

1.8.4.8 Recursive interleaving

The interleaving is applied in multi-stage manner. Figure 1.10 shows the interleaving method.

Figure 1.10 — One stage of interleaving

In the multistage interleaving, the output of this one stage of interleaving is treated as a non-protected part in the next stage. Figure 1.11 shows the example of 2 stage interleaving.

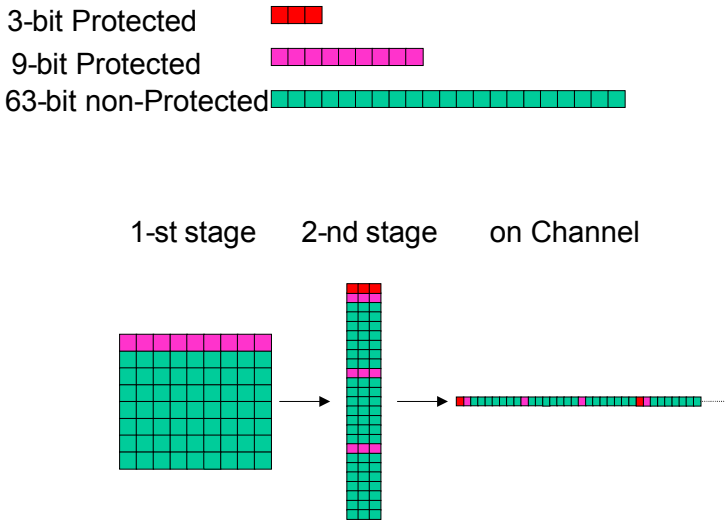


Figure 1.11 — Example of multi-stage interleaving

By choosing the width W of the interleave-matrix to be the same as the FEC code length (or the value 28 in case of SRCPC codes), the interleaving size can be optimized for all the FEC codes.

In actual case, the total number of bits for the interleaving may not allow to use such rectangular. In such case, the matrix as shown in Figure 1.12 is used.

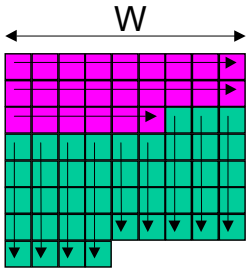


Figure 1.12 — Interleave matrix in non-rectangular case

1.8.4.8.1 Definition of recursive interleaver

Two information streams are input to this interleaver, X_i and Y_i.

X_i, 0 <= i < I_x

Y_j, 0 <= j < I_y,

where I_x and I_y is the number of bits for each input streams X_i and Y_j, respectively. X_i is set to the interleaving matrix from the top left to the bottom right, into the horizontal direction. Then Y_i is set into the rest place in vertical direction.

With the width of interleaver W, the size of interleaving matrix is shown as Figure 1.13. Where,

$D = (I_x + I_y) / W$

$d = I_x + I_y - D * W$

Where '/' indicates division by truncation.

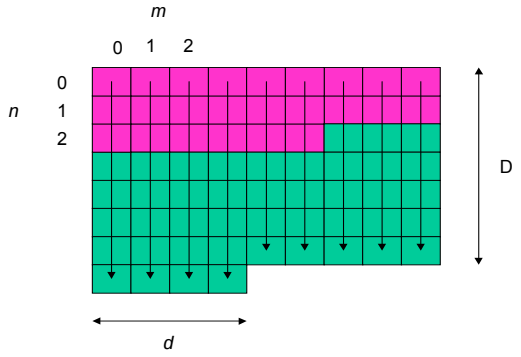


Figure 1.13 — The size of interleaving matrix

The output bitstream payload Z_k (0 < k <= I_x+I_y) is read from this matrix from top left to bottom right, column by column in horizontal direction. Thus the bit placed m-th column, n-th row (m and n starts from 0) corresponds to Z_k where:

$k = m * D + \min(m, d) + n$

In the matrix, X_i is set to

$m = i \bmod W, \quad n = i / W,$

Thus Z_k which is set by the X_i becomes:

$Z_k = X_i$, where $k = (i \bmod W) * D + \min(i \bmod W, d) + i / W$

The bits which are set with X_i in the interleaving matrix are shown as Figure 1.14 where:

$D' = l_x / W$

$d' = l_x - D' * W$

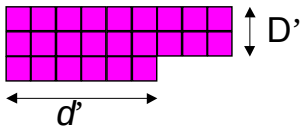


Figure 1.14 — The bits which are set with Xi in the interleaving matrix

Thus, in the m-th row, Y_j is set from the n-th row where n = D' + (m < d' ? 1 : 0) to the bottom. Thus Z_k set by Y_j is represented as follows:

```
Set j to 0;
for m = 0 to D-1 {
    for k = m * D + min(m, d) + D' + (m < d' ? 1 : 0) to (m+1) * D + min(m+1, d) - 1 {
        Z_k = Y_j;
        j ++;
    }
}
```

1.8.4.8.2 Modes of interleaving

Two modes of interleaving, mode 1 and mode 2, according to interleave_type 1 and 2, are defined in the following subclauses. Table 1.50 and Table 1.51 give an overview of the available configurations.

Table 1.50 — Width of the interleaving matrix

interleave_type	fec_type == 0 (SRCPC)	fec_type ==1/2 (SRS)
0	no interleaving	
1	28 bit	length of class
2	depends on interleave_switch (see Table 1.51)	
3	reserved	

Table 1.51 — Width of the interleaving matrix for interleave_type 2

interleave_switch	fec_type == 0 (SRCPC)	fec_type ==1/2 (SRS)
0	no interleaving	
1	length of class	length of class
2	28 bit	not permitted
3	concatenation	

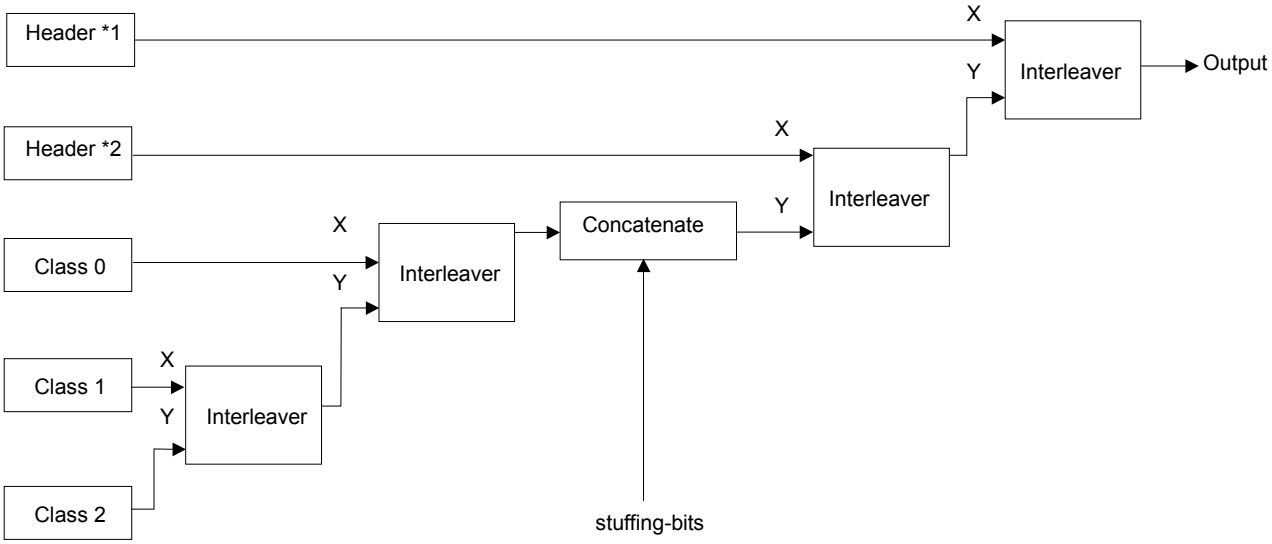
In the case of fec_type=0 (SRCPC), the interlaving is performed bitwise. In the case of fec_type == 1 or fec_type == 2 (SRS), the interleaving is performed bytewise.

1.8.4.8.2.1 Interleaving operation in mode 1

Multi-stage interleaving is processed for **ep_encoded_class** from the last class to first class, then the stuffing-bits are appended at the end of the interleaved classes. The interleaving process continues with class attribution part of ep_header() (which is class_attrib() + **class_attrib_parity**), and the pre-defined part of ep_header() (which is **choice_of_pred** + **choice_of_pred_parity**)**,** as illustrated in Figure 1.15.

The width of the interleaving matrix is chosen according to the FEC in use. In the case of SRCPC coding (fec_type == 0), the width of the interleaving matrix is 28 bit. In the case of SRS coding (fec_type == 1 or 2), the width of the interleaving matrix is equal to the length of the class in bytes. Figure 1.16 shows the interleaving scheme principle for the latter case. The bits in the class are written into the interleaving matrix byte by byte for each column.

The width of the interleaving matrix for the header parts is either equal to the length of the codeword (in bits) provided by the block code according to Table 1.48, or 28 bits if SRCPC is used.



*1 First part of Header : class_attrib() followed by class_attrib_parity
*2 Second part of Header : choice_of_pred followed by choice_of_pred_parity

Figure 1.15 — Interleaving process of mode1 specification

1.8.4.8.2.2 Interleaving operation in mode 2

In mode 2, a flag indicates whether the class is processed with interleaver, and how it is interleaved. This flag interleave_switch is signaled within the out-of-band information. The value 0 indicates the class is not processed by the interleaver. The value 1 indicates the class is interleaved by the recursive interleaver, and the length of the class is used as the width of the interleaver (either the length in bits in case of SRCPC or the length in bytes in case of SRS). The value 2 indicates the class is interleaved by the recursive interleaver, and the width is set to be equal to 28 (permitted only in case of SRCPC). The value 3 indicates the class is concatenated but not interleaved by the recursive interleaver. The interleaving operation for the ep_header is same as mode 1.

Figure 1.16 shows the interleaving scheme principle for fec_type == 1 or 2 (SRS) and interleave_switch == 1. The width is set to be the number of bytes in the class. The bits in the class are written into the interleaving matrix byte by byte for each column.

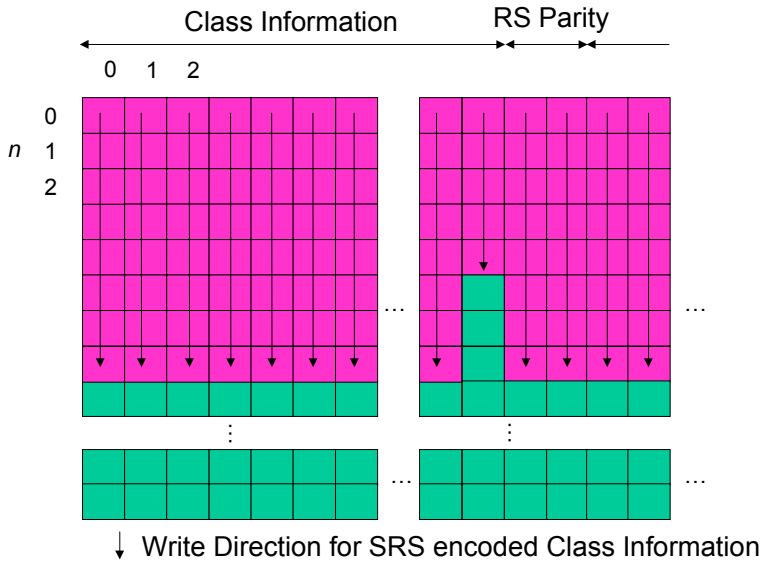


Figure 1.16 — Interleaving matrix in RS encoded class case

The interleaving process to obtain interleaved_frame_mode2 is as follows (N: number of classes):

```
    clear buffer BUF_NO /* Buffer for non-interleaved part. */
    clear buffer BUF_Y  /* Buffer for Y input in the next stage */
    for( j = 0; j < N; j++ )
       if ( class_reordered_output == 1 ) {
          k = class_output_order[choice_of_pred][j];
       } else {
          k = j;
       }
       if ( interleave_switch[choice_of_pred][k] == 3) {
       add ep_encoded_class[k] to BUF_Y;
       }
       if ( interleave_switch[choice_of_pred][k] == 0 ) {
       add ep_encoded_class[k] to BUF_NO;
       }
    }
    for ( j = N-1; j >= 0; j-- ) {
       if ( class_reordered_output == 1 ) {
          k = class_output_order[choice_of_pred][j];
       } else {
          k = j;
       }
       if ( ( interleave_switch[choice_of_pred][k] != 0 )
             && ( interleave_switch[choice_of_pred][k] != 3 ) ){
          if ( interleave_switch[choice_of_pred][k] == 1 ) {
             set the size of the interleave window to be the length of ep_encoded_class[k];
          } else if ( interleave_switch[choice_of_pred][k] == 2 ) {
             set the size of the interleave window to be 28;
          }
          input ep_encoded_class[k] into the recursive interleaver as X input;
          input BUF_Y into the recursive interleaver as Y input;
          set the output of the interleaver into BUF_Y;
       }
    }
    add BUF_NO to BUF_Y;
    if( bit_stuffing ) {
       add Nstuff stuffing-bits to BUF_Y;
    }
    input class_attrib() followed by class_attrib_parity into the recursive interleaver as X
input;
    input BUF_Y into the recursive interleaver as Y input;
```

```
    set the output of the interleaver into BUF_Y;
    input choice_of_pred followed by choice_of_pred_parity into the recursive interleaver as X
input;
    input BUF_Y into the recursive interleaver as Y input;
    set the output of the interleaver into BUF_Y;
    set BUF_Y into interleaved_frame_mode2;
```

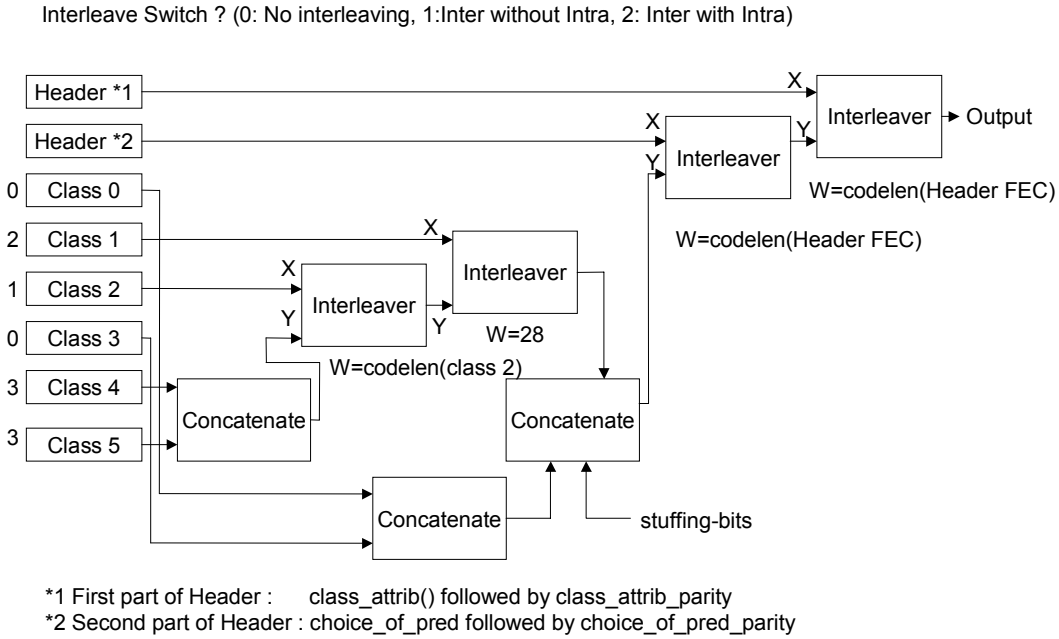


Figure 1.17 — Interleave process with class-wise control of interleaving

1.8.4.9 Class reordered output

The EP tool allows to reorder the classes such that it does not have to stick at the order provided / required by the audio codec. Figure 1.18 gives an example of the reordering process on decoder side. The class order after this reordering is signaled as **class_output_order[i][j]** in the out-of-band information. The EP decoder reorders the classes in the EP frame using i-th pre-defined set, so that the j-th class of EP frame is output as (**class_output_order[i][j]**)-th class to the audio decoder.

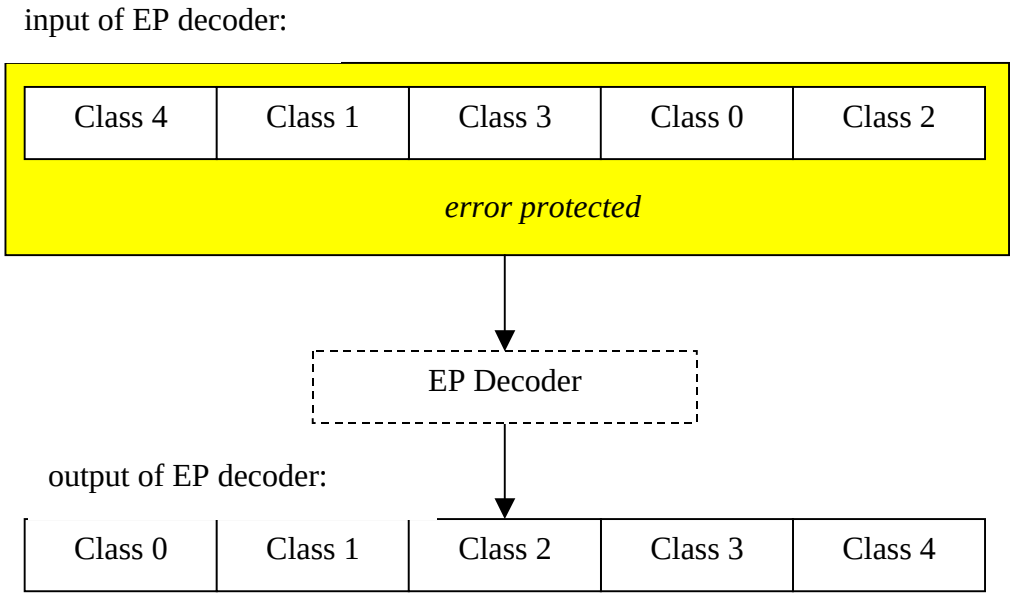


Figure 1.18 — Example for class reordered output with class_output_order 4, 1, 3, 0, 2

Annex 1.A
(informative)

Audio Interchange Formats

1.A.1 Introduction

The full capabilities and flexibility of MPEG-4 Audio, like the composition of audio scenes out of multiple audio objects, synthetic audio, and text to speech conversion, is available only if MPEG-4 Audio is used together with MPEG-4 Systems (ISO/IEC-14496-1). The interchange formats, defined in here in Annex A, only support a small subset of the capabilities of MPEG-4 Audio, by defining formats for the storage and transmission of a single mono or stereo or multi-channel audio object, very similar to the formats defined in MPEG-1 and MPEG-2.
As already stated in the introduction to subpart 1, the normative elements in MPEG-4 Audio end with the definition of the payloads (roughly equivalent to a bitstream frame in MPEG-1 and MPEG-2), and the coder configuration structures (resembling the MPEG-1/2 header information). However, there is no normative definition in MPEG-4 Audio how these elements are multiplexed, as this is required only for a limited number of applications. Nevertheless, this informative annex describes such a multiplex. However, MPEG-4 decoders are not obliged to comply to these interface formats.

1.A.2 AAC Interchange formats

1.A.3 Syntax

1.A.3.1 MPEG-2 AAC Audio_Data_Interchange_Format, ADIF

Table 1.A.1 — Syntax of adif_sequence

Syntax	No. of bits	Mnemonic

adif_sequence()
{
  adif_header();
  byte_alignment();
  raw_data_stream();
}		

Table 1.A.2 — Syntax of adif_header()

Syntax	No. of bits	Mnemonic
adif_header()		
{		
adif_id;	**32**	**bslbf**
copyright_id_present;	**1**	**bslbf**
if (copyright_id_present)		
copyright_id;	**72**	**bslbf**
original_copy;	**1**	**bslbf**
home;	**1**	**bslbf**
bitstream_type;	**1**	**bslbf**
bitrate;	**23**	**uimsbf**
num_program_config_elements;	**4**	**bslbf**
if (bitstream_type == '0') {		
adif_buffer_fullness;	**20**	**uimsbf**
}		
for (i = 0; i < num_program_config_elements + 1; i++) {		
program_config_element();		
}		
}		

Table 1.A.3 — Syntax of raw_data_stream()

Syntax	No. of bits	Mnemonic
raw_data_stream()		
{		
while (data_available()) {		
raw_data_block();		
}		
}		

1.A.3.2 Audio_Data_Transport_Stream frame, ADTS

Table 1.A.4 — Syntax of adts_sequence()

Syntax	No. of bits	Mnemonic
adts_sequence()		
{		
while (nextbits() == syncword) {		
adts_frame();		
}		
}		

Table 1.A.5 — Syntax of adts_frame()

Syntax	No. of bits	Mnemonic

adts_frame()
{
adts_fixed_header();
adts_variable_header();
if (number_of_raw_data_blocks_in_frame == 0) {
adts_error_check();
raw_data_block();
}
else {
adts_header_error_check();
for(i = 0; i <= number_of_raw_data_blocks_in_frame; i++){
raw_data_block();
adts_raw_data_block_error_check();
}
}
}		

1.A.3.2.1 Fixed Header of ADTS

Table 1.A.6 — Syntax of adts_fixed_header()

Syntax	No. of bits	Mnemonic
adts_fixed_header() {		
syncword;	**12**	**bslbf**
ID;	**1**	**bslbf**
layer;	**2**	**uimsbf**
protection_absent;	**1**	**bslbf**
profile_ObjectType;	**2**	**uimsbf**
sampling_frequency_index;	**4**	**uimsbf**
private_bit;	**1**	**bslbf**
channel_configuration;	**3**	**uimsbf**
original_copy;	**1**	**bslbf**
home;	**1**	**bslbf**
}		

1.A.3.2.2 Variable Header of ADTS

Table 1.A.7 — Syntax of adts_variable_header()

Syntax	No. of bits	Mnemonic
adts_variable_header() {		
copyright_identification_bit;	**1**	**bslbf**
copyright_identification_start;	**1**	**bslbf**
aac_frame_length;	**13**	**bslbf**
adts_buffer_fullness;	**11**	**bslbf**
number_of_raw_data_blocks_in_frame;	**2**	**uimsbf**
}		

1.A.3.2.3 Error detection

Table 1.A.8 — Syntax of adts_error_check

Syntax	No. of bits	Mnemonic

adts_error_check()
{
if (protection_absent == '0')
crc_check;
	16	**Rpchof**

}		

Table 1.A.9 — Syntax of adts_header_error_check

Syntax	No. of bits	Mnemonic

adts_header_error_check ()
{
if (protection_absent == '0') {
for (i = 1; i <= number_of_raw_data_blocks_in_frame; i++) {
raw_data_block_position(i);
}
crc_check;
}
	16
	
	16
		uimsfb
		
		rpchof

}		

Table 1.A.10 — Syntax of adts_raw_data_block_error_check()

Syntax	No. of bits	Mnemonic

adts_raw_data_block_error_check(i)
{
if (protection_absent == '0')
crc_check;
	16	**rpchof**

}		

1.A.4 Semantic

1.A.4.1 Overview

The raw_data_block() contains all data which belongs to the audio (including ancillary data). Beyond that, additional information like sampling_frequency is needed to fully describe an audio sequence. The Audio Data Interchange Format (ADIF) contains all elements that are necessary to describe a bitstream according to this standard.

For specific applications some or all of the syntax elements like those specified in the header of the ADIF, e.g. sampling_rate, may be known to the decoder by other means and hence do not appear in the bitstream.

Furthermore, additional information that varies from block to block (e.g. to enhance the parsability or error resilience) may be required. Therefore transport streams may be designed for a specific application and are not specified in this standard. However, one non-normative transport stream, called Audio Data Transport Stream (ADTS), is described. It may be used for applications in which the decoder can parse this stream.

1.A.4.2 Audio Data Interchange Format (ADIF)

The semantic of the Audio Data Interchange Format (ADIF) is specified in ISO/IEC 13818-7. The following additional definitions apply in the context of MPEG-4:

raw_data_stream() sequence of raw_data_block()'s.

program_config_element() Contains information about the configuration for one program. See subpart 4 for the definition.

raw_data_block() Defined in subpart 4.

1.A.4.3 Audio Data Transport Stream (ADTS)

The semantic of the Audio Data Transport Stream (ATDS) is specified in ISO/IEC 13818-7. The following changes apply when used in MPEG4:

raw_data_block() Defined in subpart 4.

ID MPEG identifier, set to '1' if the audio data in the ADTS stream is MPEG-2 AAC (see ISO/IEC 13818-7) and set to '0' if the audio data is MPEG-4. See also ISO/IEC 11172-3, subclause 2.4.2.3.

profile_ObjectType The interpretation of this data element depends on the value of the ID bit. If ID is equal to '1' this field holds the same information as the profile field in the ADTS stream defined in ISO/IEC 13818-7. If ID is equal to '0' this element denotes the MPEG-4 Audio Object Type (profile_ObjectType+1) according to the table defined in subclause 1.5.2.1.

Table 1.A.11 — MPEG-2 Audio profiles and MPEG-4 Audio object types

profile_ObjectType	MPEG-2 profile (ID == 1)	MPEG-4 object type (ID == 0)
0	Main profile	AAC Main
1	Low Complexity profile (LC)	AAC LC
2	Scalable Sampling Rate profile (SSR)	AAC SSR
3	(reserved)	AAC LTP

sampling_frequency_index Indicates the sampling frequency used according to the table defined in subclause 1.6.3.4. The escape value is not permitted.

channel_configuration Indicates the channel configuration used. In the case of (channel_configuration > 0), the channel configuration is given in Table 1.17. In the case of (channel_configuration == 0), the channel configuration is not specified in the header, but as follows:

MPEG-2/4 ADTS: A single program_config_element() following as first syntactic element in the first raw_data_block() after the header specifies the channel configuration. Note that the program_config_element() might not be present in each frame. An MPEG-4 ADTS decoder should not generate any output until it received a program_config_element(), while an MPEG-2 ADTS decoder may assume an implicit channel configuration.

MPEG-2 ADTS: Beside the usage of a program_config_element(), the channel configuration may be assumed to be given implicitly (see ISO/IEC13818-7) or may be known in the application.

Annex 1.B
(informative)

Error protection tool

Text file format of out-of-band information, and its example for AAC, Twin-VQ, CELP, and HVXC are presented. In addition, the example of error concealment is presented.

1.B.1 Example of out-of-band information

1.B.1.1 Example for AAC

based on the error sensitivity category assignement described within the normative part, the following error protection setup could be used, while sensitivity categories are directly mapped to classes. This example shows just a simple setup using one channel and no extension_payload().

class	length	interleaving	SRCPC puncture rate	CRC length
0	6 bit in-band field	intra-frame	8/24	6
1	12 bit in-band field	intra-frame	8/24	6
2	9 bit in-band field	inter-frame	8/8	6
3	9 bit in-band field	none	8/8	4
4	until the end	inter-frame	8/8	none

1.B.1.2 Example for Twin-VQ

This subclause describes examples of the bit assignment of UEP to the Scalable Audio Profile (TwinVQ object).

Here two encoding modes, PPC (Periodic Peak Component) - enable mode and disable mode, are described. Normally, encoder can adaptively select the PPC switch, but we force the switch always ON or always OFF in this experiment. If PPC is ON, 43 bits are assigned to quantize periodic peak components and these bits should be protected as side information.

For each mode we show bit assignment of four different bitrates, 16 kbit/s mono, 32 kbit/s stereo, 8 kbit/s + 8 kbit/s scalable mono and 16 kbit/s + 16 kbit/s stereo for each mode.

In all cases, error correction and detection tools are applied to only 10 % of bits for the side information. Remaining bits for the index of MDCT coefficients have no protection at all. As a result of these bit allocations, increase of bitrate comparing with the original source rate is around 10 % in case of PPC switch is ON, and less than 10% in case of the switch is OFF.

(A) PPC(Periodic Peak Component) enable version

- 16 kbit/s mono
  Class 1: 121 bit(fixed), SRCPC code rate 8/12, 8 bit CRC
  Class 2: 839 bit(fixed), SRCPC code rate 8/8, no CRC

- 32 kbit/s stereo
  Class 1: 238 bit(fixed), SRCPC code rate 8/12, 10 bit CRC
  Class 2: 1682 bit(fixed), SRCPC code rate 8/8, no CRC

- 8 kbit/s + 8 kbit/s scalable mono
  Class 1: 121 bit(fixed), SRCPC code rate 8/12, 8 bit CRC
  Class 2: 359 bit(fixed), SRCPC code rate 8/8, no CRC

Class 3: 72 bit(fixed), SRCPC code rate 8/12, 8 bit CRC
Class 4: 408 bit(fixed), SRCPC code rate 8/8, no CRC

- 16 kbit/s + 16 kbit/s scalable stereo
  Class 1: 238 bit(fixed), SRCPC code rate 8/12, 10 bit CRC
  Class 2: 722 bit(fixed), SRCPC code rate 8/8, no CRC
  Class 3: 146 bit(fixed), SRCPC code rate 8/12, 10 bit CRC
  Class 4: 814 bit(fixed), SRCPC code rate 8/8, no CRC

(B)PPC disable version

- 16 kbit/s mono
  Class 1: 78 bit(fixed), SRCPC code rate 8/12, 8 bit CRC
  Class 2: 882 bit(fixed), SRCPC code rate 8/8, no CRC

- 32 kbit/s stereo
  Class 1: 152 bit(fixed), SRCPC code rate 8/12, 10 bit CRC
  Class 2: 1768 bit(fixed), SRCPC code rate 8/8, no CRC

- 8 kbit/s + 8 kbit/s scalable mono
  Class 1: 78 bit(fixed), SRCPC code rate 8/12, 8 bit CRC
  Class 2: 402 bit(fixed), SRCPC code rate 8/8, no CRC
  Class 3: 72 bit(fixed), SRCPC code rate 8/12, 8 bit CRC
  Class 4: 408 bit(fixed), SRCPC code rate 8/8, no CRC

- 16 kbit/s + 16 kbit/s scalable stereo
  Class 1: 152 bit (fixed), SRCPC code rate 8/12, 10 bit CRC
  Class 2: 808 bit (fixed), SRCPC code rate 8/8, no CRC
  Class 3: 146 bit (fixed), SRCPC code rate 8/12, 10 bit CRC
  Class 4: 814 bit (fixed), SRCPC code rate 8/8, no CRC

1.B.1.3 Example for CELP

The following tables provide an overview of the number of bit that are assigned to each error sensitivity category, dependent on the configuration.

1.B.1.3.1 MPE-Narrowband Mode

Table 1.B.1 — Overview of bit assignment for MPE-narrowband

MPE-Mode	subframes	Bit/Frame	bitrate	ECR0	ECR1	ECR2	ECR3	ECR4
0	4	154	3850	6	13	20	37	78
1	4	170	4250	6	13	20	41	90
2	4	186	4650	6	13	20	45	102
3	3	147	4900	5	11	16	36	79
4	3	156	5200	5	11	16	39	85
5	3	165	5500	5	11	16	42	91
6	2	114	5700	4	9	12	29	60
7	2	120	6000	4	9	12	31	64
8	2	126	6300	4	9	12	33	68
9	2	132	6600	4	9	12	35	72
10	2	138	6900	4	9	12	37	76
11	2	142	7100	4	9	12	39	78
12	2	146	7300	4	9	12	41	80
13	4	154	7700	6	13	20	41	74
14	4	166	8300	6	13	20	45	82
15	4	174	8700	6	13	20	49	86
16	4	182	9100	6	13	20	53	90

17	4	190	9500	6	13	20	57	94
18	4	198	9900	6	13	20	61	98
19	4	206	10300	6	13	20	65	102
20	4	210	10500	6	13	20	69	102
21	4	214	10700	6	13	20	73	102
22	2	110	11000	4	9	12	33	52
23	2	114	11400	4	9	12	35	54
24	2	118	11800	4	9	12	37	56
25	2	120	12000	4	9	12	39	56
26	2	122	12200	4	9	12	41	56
27	4	186	6200	6	13	20	49	98
28	reserved							
29	reserved							
30	reserved							
31	reserved							

1.B.1.3.2 MPE-Wideband Mode

Table 1.B.2 — Overview of bit assignment for MPE-wideband

MPE-Mode	subframes	Bit/Frame	bitrate	ECR0	ECR1	ECR2	ECR3	ECR4
0	4	218	10900	17	20	27	45	109
1	4	230	11500	17	20	27	49	117
2	4	242	12100	17	20	27	53	125
3	4	254	12700	17	20	27	57	133
4	4	266	13300	17	20	27	61	141
5	4	278	13900	17	20	27	65	149
6	4	286	14300	17	20	27	69	153
7	reserved							
8	8	294	14700	17	32	43	61	141
9	8	318	15900	17	32	43	69	157
10	8	342	17100	17	32	43	77	173
11	8	358	17900	17	32	43	85	181
12	8	374	18700	17	32	43	93	189
13	8	390	19500	17	32	43	101	197
14	8	406	20300	17	32	43	109	205
15	8	422	21100	17	32	43	117	213
16	2	136	13600	17	14	19	29	57
17	2	142	14200	17	14	19	31	61
18	2	148	14800	17	14	19	33	65
19	2	154	15400	17	14	19	35	69
20	2	160	16000	17	14	19	37	73
21	2	166	16600	17	14	19	39	77
22	2	170	17000	17	14	19	41	79
23	reserved							
24	4	174	17400	17	20	27	37	73
25	4	186	18600	17	20	27	41	81
26	4	198	19800	17	20	27	45	89
27	4	206	20600	17	20	27	49	93
28	4	214	21400	17	20	27	53	97
29	4	222	22200	17	20	27	57	101
30	4	230	23000	17	20	27	61	105
31	4	238	23800	17	20	27	65	109

1.B.1.3.3 RPE-Wideband Mode

Table 1.B.3 — Characteristic parameters for wideband CELP with RPE

RPE Mode	subframes	Bit/frame	bitrate	ERC0	ERC1	ERC2	ERC3	ERC4
0	6	216	14400	40	24	34	25	93
1	4	160	16000	32	18	26	21	63
2	8	280	18667	48	30	42	29	131
3	10	338	22533	56	36	50	33	163

1.B.1.4 Example for HVXC

- 2 kbit/s source coder
  Class 1: 22 bit (fixed), SRCPC code rate 8/16, 6 bit CRC
  Class 2: 4 bit (fixed), SRCPC code rate 8/8,1 bit CRC
  Class 3: 4 bit (fixed), SRCPC code rate 8/8,1 bit CRC
  Class 4: 10 bit (fixed), SRCPC code rate 8/8, no CRC

- 4 kbit/s source coder
  Class 1: 33 bit (fixed), SRCPC code rate 8/16, 6 bit CRC
  Class 2: 22 bit (fixed), SRCPC code rate 8/8, 6 bit CRC
  Class 3: 4 bit (fixed), SRCPC code rate 8/8, 1 bit CRC
  Class 4: 4 bit (fixed), SRCPC code rate 8/8, 1 bit CRC
  Class 5: 17 bit (fixed), SRCPC code rate 8/8, no CRC

1.B.1.5 Example for ER BSAC

This subclause describes examples of the bit assignment of Unequal Error Protection (UEP) to the ER BSAC Object Type.

The lower error sensitivity category (ESC) described in subpart 4, subclause 4.5.2.6.3 indicates the class with the higher error sensitivity, whereas the higher ESC indicates the class with the lower sensitivity. Based on the error sesitivity category of the BSAC syntax, the following error protection setup example could be used, This example shows just a simple setup where sensitivity categories are directly mapped to classes.

Class	category	length	interleaving	SRCPC puncture rate	CRC length
0	0	9 bit in-band field	intra-frame	8/24	6
1	others	11 bit in-band field	no	8/8	none

In this example, error correction and detection tools are applied to only the general side information. Remaining bits for the index of MDCT coefficients have no protection at all. As a result of these bit allocations, increase of bitrate comparing with the original source rate is around 10 %.

And, the predefined set for UEP can be set up in addition to the UPE class setup as follows :

1.B.2 Example of error concealment

The error concealment tool is an optional decoder tool to reduce the quality degradation of decoded signals when the decoder input bitstream payload is affected by errors, such as bitstream payload transmission error. This is especially valid in applying the MPEG-4 Audio tools to radio applications. The error detection and decision method to replace a frame is not defined in this subclause and decided based on each application.

1.B.2.1 General

Detailed examples of the out-of-band information for the individual codec types are provided in ISO/IEC14496-5 in conjunction with the ep-tool software. Further examples are given in ISO/IEC 14496-4 by means of conformance test sequences with epConfig=2 and epConfig=3.

1.B.2.2 Example for CELP

1.B.2.2.1 Overview of the error concealment tool

The error concealment tool is used with the MPEG-4 CELP decoder described in ISO/IEC 14496-3. This tool reduces unpleasant noise when the MPEG-4 CELP decodes speech from erroneous input frame data. It also enables the MPEG-4 CELP to decode speech even if the input frame data is lost.

The tool has two operating modes|: a Bit Error (BE) mode and a Frame Erasure (FE) mode. The mode is switched based on the availability of frame data at the decoder. When frame data is available (the BE mode), decoding is done using the frame data received in the past and the usable subpart of the current frame data. When frame data is not available (the FE mode), the decoder generates the speech using only the past frame data.

This tool operates according to a flag (the BF_flag) that indicates whether the frame data is complete (BF_flag=0), or is damaged by corruption and/or lost (BF_flag=1). The flag is usually given by the channel coder or the transmission system.

This tool operates in Coding Mode II (subclause 3.1.2.1 of ISO/IEC 14496-3:2001), which has narrow band, wideband and band width scalable modes that use Multi-Pulse Excitation at sampling rates of 8 and 16 kHz.

1.B.2.2.2 Definitions

BE: Bit Error
BWS: BandWidth Scalable
FE: Frame Erasure
LP: Linear Prediction
LSP: Line Spectral Pair
MPE: Multi-Pulse Excitation
NB: Narrow Band
RMS: Root Mean Square (the frame energy)
WB: WideBand

1.B.2.2.3 Helping variables

frame_size: the number of samples in a frame
g_ac: the adaptive codebook gain
g_ec: the MPE gain
lpc_order: the order of LP
signal_mode: the speech mode
signal_mode_pre: the speech mode of the previous frame

1.B.2.2.4 Specifications of the error concealment tool

The error concealment tool operates based on the transition model with six states depicted in Figure 1.B.1. The state indicates the quality of the transmission channel. The bigger the state number is, the worse the channel quality is. Each state has a different concealment operation. The initial state in decoding is State 0 and the state is transited based on the *BF_flag*. Each concealment operation is described in the following subclauses.

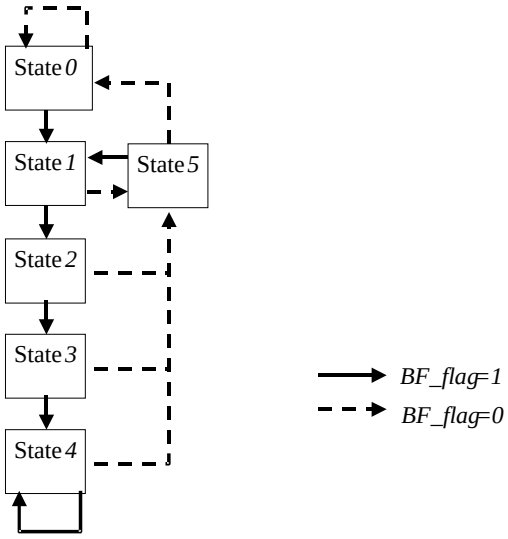


Figure 1.B.1 — State transition model for controlling the error concealment

1.B.2.2.4.1 Operations in States *0* and *5*

The decoding process is identical to that of the MPEG-4 CELP decoder with the following exceptions regarding the adaptive codebook and the excitation codebook gains:

In State *0* following State *5*, and in State *5*:

(1) for the first *80* samples in and after the frame where the *BF_flag* is changed from *1* to *0*, the gains *g_ac* and *g_ec* are calculated from the gains *g_ac'* and *g_ec'* decoded from the current frame data as follows:

```
if (g_ac > 1.0) {
        g_ec = g_ec'/ g_ac';
        g_ac=1.0;
}
```

(2) for the *160* samples that follow the first *80* samples, the gains are calculated as:

```
if(g_ac > 1.0){
        g_ec = g _ec'*(g_ac'+1.0) /g_ac'/2.0 ;
        g_ac = (g_ac'+1.0)/2.0;
},
```

where these operations continue in, at most, four subframes.

1.B.2.2.4.2 Operations in States *1*, *2*, *3* and *4*

The decoding process is identical to that of the MPEG-4 CELP decoder with the exceptions described in the following subclauses.

1.B.2.2.4.2.1 Speech mode

1.B.2.2.4.2.1.1 FE mode

The speech mode (*signal_mode*) is decoded from the previous frame data.

1.B.2.2.4.2.1.2 BE mode

The speech mode (*singal_mode*) is decoded from the current frame data for *signal _mode_pre=0* or *1*. Otherwise, the mode is decoded from the previous frame data.

1.B.2.2.4.2.2 Multi-Pulse Excitation (MPE)

1.B.2.2.4.2.2.1 FE mode

The MPE is decoded from the randomly generated frame data.

1.B.2.2.4.2.2.2 BE mode

The MPE is decoded from the frame data received in the current frame.

1.B.2.2.4.2.3 RMS

For State *1* through *K*, the RMS in the last subframe of the previous frame is used after being attenuated in the first subframe of the current frame. In the following subframes, the RMS in the previous subframe is used after being attenuated. The attenuation level P_{att} depends on the state as follows:

$$P_{att} = \begin{cases} 0.4\ dB & for\ \text{State}\ 1,\ldots,K \\ 1.2\ dB & for\ \text{State}\ K+1,\ldots \end{cases},$$

where K is the smaller number of *4* and K_0, and K_0 is the maximum integer satisfying $frame\_size \times K_0 \le 320$.

1.B.2.2.4.2.4 LSP

The LSPs decoded in the previous frame are used. In the BWS mode, the LSP codevectors should be buffered for the interframe prediction described in subclause 3.5.6.3.3 of ISO/IEC 14496-3:2001. However, when the frame data is corrupted or lost, the correct codevector can not be obtained. Therefore, the buffered codevector (*blsp*[*0*][]) is estimated from the LSP (*qlsp_pre*[]) in the previous frame, the predicted LSP (*vec_hat*[]) and the prediction coefficient (*cb*[*0*][]) in the current frame as follows:

```
for (i = 0; i < lpc_order; i++) {
    blsp[0] [i] = (qlsp_pre[i] - vec_hat[i])/cb[0] [i];
}
```

1.B.2.2.4.2.5 Delay of the adaptive codebook

1.B.2.2.4.2.5.1 FE mode

All the delays of the adaptive codebook are decoded from the delay index received in the last subframe of the previous frame.

1.B.2.2.4.2.5.2 BE mode

(1) When *signal_mode_pre=0*, the delays are decoded from the current frame data.

(2) When *signal_mode_pre*=1, and if the maximum difference between the delay indices of the adjacent subframes in the frame are less than *10*, the delays are decoded from the delay indices in the current frame. In each subframe where the difference in the delay indices between the current and the previous subframes is equal to or greater than *10*, the delay is decoded from the index of the previous subframe.

(3) When *signal_mode_pre= 2* or *3*, the delay is decoded from the index in the last subframe of the previous frame.

1.B.2.2.4.2.6 Gains

1.B.2.2.4.2.6.1 Index operation

1.B.2.2.4.2.6.1.1 FE mode

All the gains have the same value, which is decoded from the gain index received in the last subframe of the previous frame.

1.B.2.2.4.2.6.1.2 BE mode

The gains are decoded from current frame data.

1.B.2.2.4.2.6.2 Adjustment operation

1.B.2.2.4.2.6.2.1 FE mode

(1) When *signal_mode_pre=0*, the gains *g_ac'* and *g_ec'* are decoded from the current frame data. The gains *g_ac* and *g_ec* are then obtained by multiplying *g_ac'* by *0.5* and *g_ec'* by *X,* respectively. *X* satisfies the following equation and is calculated in each subframe:

(*g_ac'*g_ac'*)A+(*g_ec'*g_ec'*)B=((0.5*g_ac')* *(0.5*g_ac'*))A+((*X*g_ec')* *(X*g_ec'))B*

where

A = $norm_{ac} \times norm_{ac}$

B = $norm_{ec} \times norm_{ec}$.

$norm_{ac}$ and $norm_{ec}$ are the respective RMS values of the adaptive and the excitation codevectors.

(2) When *signal_mode_pre=1*, the gains are decoded from the current frame data.

(3) When *signal_mode_pre=2* or *3*, gains for the first *320* samples in or after the frame where the *BF_flag* is changed from *0* to *1,* are calculated as:

g_ac = $0.95 \times 10^{(-0.4/20)}$

g_ec = $X \times 10^{(-0.4/20)}$.

After the first *320* samples,

g_ac = $0.95 \times 10^{(-1.2/20)}$

g_ec = $X \times 10^{(-1.2/20)}$,

where *X* = $0.05 \times \frac{norm_{ac}}{norm_{ec}}$.

1.B.2.2.4.2.6.2.2 BE mode

(1) When *signal_mode_pre=0* or *1*, the gains are calculated using the gains *g_ac'* and *g_ec'* decoded from the current frame data and the gains *g_ac_pre* and *g_ec_pre* of the previous subframe so that the calculated gains fall in a normal range and generate no unpleasant noise as follows:

```
if (g_ac > 1.2589) {
    g_ec = g_ec' * 1.2589/g_ac';
    g_ac = 1.2589;
}
if (g_ec > 1.2589*g_ec_pre) {
    g_a c= g_ac'* 1.2589*g_ec_pre/g_ec' ;
g_ec  = g_ec_pre*1.2589;
}
if (signal_mode = 1 &  g_ac_pre < 1.2589 & g_ac < g_ac_pre*0.7943) {
    g_ac = g_ac_pre*0.7943;
    g_ec = g_ec_pre*0.7943;
}.
```

(2) When *signal_mode_pre=2* or *3*, the operation is identical to that for *signal_mode_pre=2* or *3* in the FE mode.

1.B.2.3 Error concealment for the silence compression tool

In frames where the bitstream payload received at the decoder is corrupted or lost due to transmission bit errors, error concealment is performed. When the received TX_flag is *1*, the decoding process is identical to that of the error concealment for the MPEG-4 CELP. For TX_flag=*0, 2* or *3*, the decoding process for the TX_flag=*0* is used.

1.B.3 Example of EP tool setting and error concealment for HVXC

This subclause describes one example of the implementation of EP (Error Protection) tool and error concealment method for HVXC. Some of perceptually important bits are protected by FEC (forward error correction) scheme and some are checked by CRC to judge whether or not erroneous bits are included. When CRC error occures, error concealment is executed to reduce perceptible degradation.

It should be noted that error correction method and EP tool setting, error concealment algorithm described below are one example, and they should be modified depending on the actual channel conditions.

1.B.3.1 Definitions

--- 2/4 kbit/s common parameters ---

LSP1	LSP index 1	(5 bit)
LSP2	LSP index 2	(7 bit)
LSP3	LSP index 3	(5 bit)
LSP4	LSP index 4	(1 bit)
VUV	voiced/unvoiced flag	(2 bit)
Pitch	pitch parameter	(7 bit)
SE_shape1	spectrum index 0	(4 bit)
SE_shape2	spectrum index 1	(4 bit)
SE_gain	spectrum gain index	(5 bit)
VX_shape1[0]	stochastic codebook index 0	(6 bit)
VX_shape1[1]	stochastic codebook index 1	(6 bit)
VX_gain1[0]	gain codebook index 0	(4 bit)
VX_gain1[1]	gain codebook index 1	(4 bit)

--- only 4 kbit/s parameters ---

LSP5	LSP index 5	(8 bit)
SE_shape3	4k spectrum index 0	(7 bit)
SE_shape4	4k spectrum index 1	(10 bit)

SE_shape5	4k spectrum index 2	(9 bit)
SE_shape6	4k spectrum index 3	(6 bit)
VX_shape2[0]	4k stochastic codebook index 0	(5 bit)
VX_shape2[1]	4k stochastic codebook index 1	(5 bit)
VX_shape2[2]	4k stochastic codebook index 2	(5 bit)
VX_shape2[3]	4k stochastic codebook index 3	(5 bit)
VX_gain2[0]	4k gain codebook index 0	(3 bit)
VX_gain2[1]	4k gain codebook index 1	(3 bit)
VX_gain2[2]	4k gain codebook index 2	(3 bit)
VX_gain2[3]	4k gain codebook index 3	(3 bit)

1.B.3.2 Channel coding

1.B.3.2.1 Protected bit selection

According to the sensitivity of bits, encoded bits are classified to several classes. The number of bits for each class is shown in the Table 1.B.4, Table 1.B.5 (2 kbit/s), Table 1.B.6 and Table 1.B.7 (4 kbit/s). As an example, bitrate setting of 3.5 kbit/s(for 2 kbit/s) and 6.2 kbit/s (for 4 kbit/s) are shown. In these cases, two source coder frames are processed as one set. Suffix “p” means parameters of the “previous” frame, and “c” means those of the “current” frame.

For 3.5 kbit/s mode, 6 classes are used. CRC check is applied for class I, II, III, IV, and V bits. Class VI bits are not checked by CRC.

The Table 1.B.4 below shows protected/unprotected(class I…VI) bit assignment in the case where both previous and current frames are voiced.

Table 1.B.4 — Number of protected/unprotected bits at 3.5 kbit/s (voiced frame)

para-meters	voiced frame						
	class I bits	**class II bits**	**class III bits**	**class IV bits**	**class V bits**	**class VI bits**	**total**
LSP1p/c	5/5	-	-	-	-	-	**10**
LSP2p/c	2/2	-	-	-	-	5/5	**14**
LSP3p/c	1/1	-	-	-	-	4/4	**10**
LSP4p/c	1/1	-	-	-	-	-	**2**
VUVp/c	2/2	-	-	-	-	**-**	**4**
Pitchp/c	6/6	-	-	-	-	1/1	**14**
SE_gainp/c	5/5	-	-	-	-	-	**10**
SE_shape1p	-	4	-	-	-	-	**4**
SE_shape1c	-	-	-	4	-	-	**4**
SE_shape2p	-	-	4	-	-	-	**4**
SE_shape2c	-	-	-	-	4	-	**4**
total	**44**	**4**	**4**	**4**	**4**	**20**	**80**

The Table 1.B.5 shows protected/unprotected(class I…VI) bit assignment in the case where both previous and current frames are unvoiced.

Table 1.B.5 — Number of protected/unprotected bits at 3.5 kbit/s (unvoiced frame)

para-meters	unvoiced frame						
	class I bits	class II bits	class III bits	class IV bits	Class V bits	class VI bits	total
LSP1p/c	5/5	-	-	-	-	-	**10**
LSP2p/c	4/4	-	-	-	-	3/3	**14**
LSP3p/c	2/2	-	-	-	-	3/3	**10**
LSP4p/c	1/1	-	-	-	-	-	**2**
VUVp/c	2/2	-	-	-	-	-	**4**
VX_gain1[0]p/c	4/4	-	-	-	-	-	**8**
VX_gain1[1]p/c	4/4	-	-	-	-	-	**8**
VX_shape1[0]p/	-	-	-	-	-	6/6	**12**
VX_shape1[1]p/	-	-	-	-	-	6/6	**12**
total	**44**	**0**	**0**	**0**	**0**	**36**	**80**

When the previous frame is unvoiced and the current frame is voiced, or when the previous frame is voiced and the current frame is unvoiced, the same protected/unprotected bit assignment rules as shown above are used.

For 6.2 kbit/s mode, 7 classes are used. CRC check is applied for class I, II, III, IV, V, and VI bits. Class VII bits are not checked by CRC.

The Table 1.B.6 shows protected/unprotected(class I…VII) bit assignment in the case where both previous and current frames are voiced.

Table 1.B.6 — Number of protected/unprotected bits at 6.2 kbit/s (voiced sound)

Para-meters	voiced sound							
	class I bits	class II bits	class III bits	class IV bits	class V bits	class VI bits	class VII bits	total
LSP1p/c	5/5	-	-	-	-	-	-	**10**
LSP2p/c	4/4	-	-	-	-	-	3/3	**14**
LSP3p/c	1/1	-	-	-	-	-	4/4	**10**
LSP4p/c	1/1	-	-	-	-	-	-	**2**
LSP5p/c	1/1	-	-	-	-	-	7/7	**16**
VUVp/c	2/2	-	-	-	-	-	-	**4**
Pitchp/c	6/6	-	-	-	-	-	1/1	**14**
SE_gainp/c	5/5	-	-	-	-	-	-	**10**
SE_shape1p	-	-	4	-	-	-	-	**4**
SE_shape1c	-	-	-	-	4	-	-	**4**
SE_shape2p	-	-	-	4	-	-	-	**4**
SE_shape2c	-	-	-	-	-	4	-	**4**
SE_shape3p/c	5/5	-	-	-	-	-	2/2	**14**
SE_shape4p/c	1/1	9/9	-	-	-	-	-	**20**
SE_shape5p/c	1/1	8/8	-	-	-	-	-	**18**
SE_shape6p/c	1/1	5/5	-	-	-	-	-	**12**
Total	**66**	**44**	**4**	**4**	**4**	**4**	**34**	**160**

The Table 1.B.7 below shows protected/unprotected(class I…VII) bit assignment in the case where both previous and current frames are unvoiced.

Table 1.B.7 — Number of protected/unprotected bits at 6.2 kbit/s (unvoiced sound)

Para-meters	unvoiced sound							
	class I bits	class II bits	class III bits	class IV bits	class V bits	class VI bits	class VII bits	total
LSP1p/c	5/5	-	-	-	-	-	-	**10**
LSP2p/c	4/4	-	-	-	-	-	3/3	**14**
LSP3p/c	1/1	-	-	-	-	-	4/4	**10**
LSP4p/c	1/1	-	-	-	-	-	-	**2**
LSP5p/c	1/1	-	-	-	-	-	7/7	**16**
VUVp/c	2/2	-	-	-	-	-	-	**4**
VX_gain1[0]p/c	4/4	-	-	-	-	-	-	**8**
VX_gain1[1]p/c	4/4	-	-	-	-	-	-	**8**
VX_shape1[0]p/	-	-	-	-	-	-	6/6	**12**
VX_shape1[1]p/	-	-	-	-	-	-	6/6	**12**
VX_gain2[0]p/c	3/3	-	-	-	-	-	-	**6**
VX_gain2[1]p/c	3/3	-	-	-	-	-	-	**6**
VX_gain2[2]p/c	3/3	-	-	-	-	-	-	**6**
VX_gain2[3]p/c	2/2	-	-	-	-	-	1/1	**6**
VX_shape2[0]p/	-	-	-	-	-	-	5/5	**10**
VX_shape2[1]p/	-	-	-	-	-	-	5/5	**10**
VX_shape2[2]p/	-	-	-	-	-	-	5/5	**10**
VX_shape2[3]p/	-	-	-	-	-	-	5/5	**10**
total	**66**	**0**	**0**	**0**	**0**	**0**	**94**	**160**

When the previous frame is unvoiced and the current frame is voiced, or when the previous frame is voiced and the current frame is unvoiced, the same protected/unprotected bit assignment rules as shown above are used.

The bit order for UEP input is shown from Table 1.B.8 through Table 1.B.11 (for 2 kbit/s), and from Table 1.B.12 through Table 1.B.15 (for 4 kbit/s). These tables show the bit order for each of the combinations of V/UV condition of 2 frames. For example, if previous frame is voiced and current frame is voiced at 2 kbit/s mode, Table 1.B.8 is used. The bit order is arranged according to the error sensitivity. The column "Bit" denotes the bit index of the parameter. "0" means LSB.

Table 1.B.8 — Bit Order for 2 kbit/s (voiced frame -- voiced frame)

voiced frame – voiced frame								
No.	**Item**	**Bit**	**No.**	**Item**	**Bit**	**No.**	**Item**	**Bit**
Class I Bit			**28**	SE_gainc	1	**54**	SE_shape1c	1
0	VUVp	1	**29**	SE_gainc	0	**55**	SE_shape1c	0
1	VUVp	0	**30**	LSP1c	4	**Class V Bit**		
2	LSP4p	0	**31**	LSP1c	3	**56**	SE_shape2c	3
3	SE_gainp	4	**32**	LSP1c	2	**57**	SE_shape2c	2
4	SE_gainp	3	**33**	LSP1c	1	**58**	SE_shape2c	1
5	SE_gainp	2	**34**	LSP1c	0	**59**	SE_shape2c	0
6	SE_gainp	1	**35**	Pitchc	6	**Class VI Bit**		
7	SE_gainp	0	**36**	Pitchc	5	**60**	LSP2p	4

8	LSP1p	4	**37**	Pitchc	4	**61**	LSP2p	3
9	LSP1p	3	**38**	Pitchc	3	**62**	LSP2p	2
10	LSP1p	2	**39**	Pitchc	2	**63**	LSP2p	1
11	LSP1p	1	**40**	Pitchc	1	**64**	LSP2p	0
12	LSP1p	0	**41**	LSP2c	6	**65**	LSP3p	3
13	Pitchp	6	**42**	LSP3c	4	**66**	LSP3p	2
14	Pitchp	5	**43**	LSP2c	5	**67**	LSP3p	1
15	Pitchp	4	**Class II Bit**			**68**	LSP3p	0
16	Pitchp	3	**44**	SE_shape1p	3	**69**	Pitchp	0
17	Pitchp	2	**45**	SE_shape1p	2	**70**	LSP2c	4
18	Pitchp	1	**46**	SE_shape1p	1	**71**	LSP2c	3
19	LSP2p	6	**47**	SE_shape1p	0	**72**	LSP2c	2
20	LSP3p	4	**Class III Bit**			**73**	LSP2c	1
21	LSP2p	5	**48**	SE_shape2p	3	**74**	LSP2c	0
22	VUVc	1	**49**	SE_shape2p	2	**75**	LSP3c	3
23	VUVc	0	**50**	SE_shape2p	1	**76**	LSP3c	2
24	LSP4c	0	**51**	SE_shape2p	0	**77**	LSP3c	1
25	SE_gainc	4	**Class IV Bit**			**78**	LSP3c	0
26	SE_gainc	3	**52**	SE_shape1c	3	**79**	Pitchc	0
27	SE_gainc	2	**53**	SE_shape1c	2			

Table 1.B.9 — Bit Order for 2 kbit/s (voiced frame – unvoiced frame)

voiced frame – unvoiced frame								
No.	**Item**	**Bit**	**No.**	**Item**	**Bit**	**No.**	**Item**	**Bit**
Class I Bit			**28**	VX_gain1[0]c	0	**54**	LSP2c	0
0	VUVp	1	**29**	VX_gain1[1]c	3	**55**	LSP3c	2
1	VUVp	0	**30**	VX_gain1[1]c	2	**Class V Bit**		
2	LSP4p	0	**31**	VX_gain1[1]c	1	**56**	LSP3c	1
3	SE_gainp	4	**32**	VX_gain1[1]c	0	**57**	LSP3c	0
4	SE_gainp	3	**33**	LSP1c	4	**58**	VX_shape1[0]c	5
5	SE_gainp	2	**34**	LSP1c	3	**59**	VX_shape1[0]c	4
6	SE_gainp	1	**35**	LSP1c	2	**Class VI Bit**		
7	SE_gainp	0	**36**	LSP1c	1	**60**	LSP2p	4
8	LSP1p	4	**37**	LSP1c	0	**61**	LSP2p	3
9	LSP1p	3	**38**	LSP2c	6	**62**	LSP2p	2
10	LSP1p	2	**39**	LSP2c	5	**63**	LSP2p	1
11	LSP1p	1	**40**	LSP2c	4	**64**	LSP2p	0
12	LSP1p	0	**41**	LSP2c	3	**65**	LSP3p	3
13	Pitchp	6	**42**	LSP3c	4	**66**	LSP3p	2
14	Pitchp	5	**43**	LSP3c	3	**67**	LSP3p	1
15	Pitchp	4	**Class II Bit**			**68**	LSP3p	0
16	Pitchp	3	**44**	SE_shape1p	3	**69**	Pitchp	0
17	Pitchp	2	**45**	SE_shape1p	2	**70**	VX_shape1[0]c	3
18	Pitchp	1	**46**	SE_shape1p	1	**71**	VX_shape1[0]c	2
19	LSP2p	6	**47**	SE_shape1p	0	**72**	VX_shape1[0]c	1
20	LSP3p	4	**Class III Bit**			**73**	VX_shape1[0]c	0
21	LSP2p	5	**48**	SE_shape2p	3	**74**	VX_shape1[1]c	5
22	VUVc	1	**49**	SE_shape2p	2	**75**	VX_shape1[1]c	4

23	VUVc	0	**50**	SE_shape2p	1	**76**	VX_shape1[1]c	3
24	LSP4c	0	**51**	SE_shape2p	0	**77**	VX_shape1[1]c	2
25	VX_gain1[0]c	3	**Class IV Bit**			**78**	VX_shape1[1]c	1
26	VX_gain1[0]c	2	**52**	LSP2c	2	**79**	VX_shape1[1]c	0
27	VX_gain1[0]c	1	**53**	LSP2c	1			

Table 1.B.10 — Bit Order for 2 kbit/s (unvoiced frame – voiced frame)

unvoiced frame – unvoiced frame								
No.	**Item**	**Bit**	**No.**	**Item**	**Bit**	**No.**	**Item**	**Bit**
Class I Bit			**28**	SE_gainc	1	**54**	SE_shape1c	1
0	VUVp	1	**29**	SE_gainc	0	**55**	SE_shape1c	0
1	VUVp	0	**30**	LSP1c	4	**Class V Bit**		
2	LSP4p	0	**31**	LSP1c	3	**56**	SE_shape2c	3
3	VX_gain1[0]p	3	**32**	LSP1c	2	**57**	SE_shape2c	2
4	VX_gain1[0]p	2	**33**	LSP1c	1	**58**	SE_shape2c	1
5	VX_gain1[0]p	1	**34**	LSP1c	0	**59**	SE_shape2c	0
6	VX_gain1[0]p	0	**35**	Pitchc	6	**Class VI Bit**		
7	VX_gain1[1]p	3	**36**	Pitchc	5	**60**	VX_shape1[0]p	3
8	VX_gain1[1]p	2	**37**	Pitchc	4	**61**	VX_shape1[0]p	2
9	VX_gain1[1]p	1	**38**	Pitchc	3	**62**	VX_shape1[0]p	1
10	VX_gain1[1]p	0	**39**	Pitchc	2	**63**	VX_shape1[0]p	0
11	LSP1p	4	**40**	Pitchc	1	**64**	VX_shape1[1]p	5
12	LSP1p	3	**41**	LSP2c	6	**65**	VX_shape1[1]p	4
13	LSP1p	2	**42**	LSP3c	4	**66**	VX_shape1[1]p	3
14	LSP1p	1	**43**	LSP2c	5	**67**	VX_shape1[1]p	2
15	LSP1p	0	**Class II Bit**			**68**	VX_shape1[1]p	1
16	LSP2p	6	**44**	LSP2p	2	**69**	VX_shape1[1]p	0
17	LSP2p	5	**45**	LSP2p	1	**70**	LSP2c	4
18	LSP2p	4	**46**	LSP2p	0	**71**	LSP2c	3
19	LSP2p	3	**47**	LSP3p	2	**72**	LSP2c	2
20	LSP3p	4	**Class III Bit**			**73**	LSP2c	1
21	LSP3p	3	**48**	LSP3p	1	**74**	LSP2c	0
22	VUVc	1	**49**	LSP3p	0	**75**	LSP3c	3
23	VUVc	0	**50**	VX_shape1[0]p	5	**76**	LSP3c	2
24	LSP4c	0	**51**	VX_shape1[0]p	4	**77**	LSP3c	1
25	SE_gainc	4	**Class IV Bit**			**78**	LSP3c	0
26	SE_gainc	3	**52**	SE_shape1c	3	**79**	Pitchc	0
27	SE_gainc	2	**53**	SE_shape1c	2			

Table 1.B.11 — Bit Order for 2 kbit/s (unvoiced frame – unvoiced frame)

unvoiced frame – unvoiced frame								
No.	**Item**	**Bit**	**No.**	**Item**	**Bit**	**No.**	**Item**	**Bit**
Class I Bit			**28**	VX_gain1[0]c	0	**54**	LSP2c	0
0	VUVp	1	**29**	VX_gain1[1]c	3	**55**	LSP3c	2
1	VUVp	0	**30**	VX_gain1[1]c	2	**Class V Bit**		
2	LSP4p	0	**31**	VX_gain1[1]c	1	**56**	LSP3c	1

3	VX_gain1[0]p	3	**32**	VX_gain1[1]c	0	**57**	LSP3c	0
4	VX_gain1[0]p	2	**33**	LSP1c	4	**58**	VX_shape1[0]c	5
5	VX_gain1[0]p	1	**34**	LSP1c	3	**59**	VX_shape1[0]c	4
6	VX_gain1[0]p	0	**35**	LSP1c	2	**Class VI Bit**		
7	VX_gain1[1]p	3	**36**	LSP1c	1	**60**	VX_shape1[0]p	3
8	VX_gain1[1]p	2	**37**	LSP1c	0	**61**	VX_shape1[0]p	2
9	VX_gain1[1]p	1	**38**	LSP2c	6	**62**	VX_shape1[0]p	1
10	VX_gain1[1]p	0	**39**	LSP2c	5	**63**	VX_shape1[0]p	0
11	LSP1p	4	**40**	LSP2c	4	**64**	VX_shape1[1]p	5
12	LSP1p	3	**41**	LSP2c	3	**65**	VX_shape1[1]p	4
13	LSP1p	2	**42**	LSP3c	4	**66**	VX_shape1[1]p	3
14	LSP1p	1	**43**	LSP3c	3	**67**	VX_shape1[1]p	2
15	LSP1p	0	**Class II Bit**			**68**	VX_shape1[1]p	1
16	LSP2p	6	**44**	LSP2p	2	**69**	VX_shape1[1]p	0
17	LSP2p	5	**45**	LSP2p	1	**70**	VX_shape1[0]c	3
18	LSP2p	4	**46**	LSP2p	0	**71**	VX_shape1[0]c	2
19	LSP2p	3	**47**	LSP3p	2	**72**	VX_shape1[0]c	1
20	LSP3p	4	**Class III Bit**			**73**	VX_shape1[0]c	0
21	LSP3p	3	**48**	LSP3p	1	**74**	VX_shape1[1]c	5
22	VUVc	1	**49**	LSP3p	0	**75**	VX_shape1[1]c	4
23	VUVc	0	**50**	VX_shape1[0]p	5	**76**	VX_shape1[1]c	3
24	LSP4c	0	**51**	VX_shape1[0]p	4	**77**	VX_shape1[1]c	2
25	VX_gain1[0]c	3	**Class IV Bit**			**78**	VX_shape1[1]c	1
26	VX_gain1[0]c	2	**52**	LSP2c	2	**79**	VX_shape1[1]c	0
27	VX_gain1[0]c	1	**53**	LSP2c	1			

Table 1.B.12 — Bit Order for 4 kbit/s (voiced frame – voiced frame)

voiced frame – voiced frame								
No.	**Item**	**Bit**	**No.**	**Item**	**Bit**	**No.**	**Item**	**Bit**
Class 1bit			**55**	LSP2c	3	**Class III Bit**		
0	VUVp	1	**56**	SE_shape3c	6	**110**	SE_shape1p	3
1	VUVp	0	**57**	SE_shape3c	5	**111**	SE_shape1p	2
2	LSP4p	0	**58**	SE_shape3c	4	**112**	SE_shape1p	1
3	SE_gainp	4	**59**	SE_shape3c	3	**113**	SE_shape1p	0
4	SE_gainp	3	**60**	SE_shape3c	2	**Class IV Bit**		
5	SE_gainp	2	**61**	LSP3c	4	**114**	SE_shape2p	3
6	SE_gainp	1	**62**	LSP5c	7	**115**	SE_shape2p	2
7	SE_gainp	0	**63**	SE_shape4c	9	**116**	SE_shape2p	1
8	LSP1p	4	**64**	SE_shape5c	8	**117**	SE_shape2p	0
9	LSP1p	3	**65**	SE_shape6c	5	**Class V Bit**		
10	LSP1p	2	**Class II Bit**			**118**	SE_shape1c	3
11	LSP1p	1	**66**	SE_shape4p	8	**119**	SE_shape1c	2
12	LSP1p	0	**67**	SE_shape4p	7	**120**	SE_shape1c	1
13	Pitchp	6	**68**	SE_shape4p	6	**121**	SE_shape1c	0
14	Pitchp	5	**69**	SE_shape4p	5	**Class VI Bit**		
15	Pitchp	4	**70**	SE_shape4p	4	**122**	SE_shape2c	3
16	Pitchp	3	**71**	SE_shape4p	3	**123**	SE_shape2c	2

17	Pitchp	2	**72**	SE_shape4p	2	**124**	SE_shape2c	1
18	Pitchp	1	**73**	SE_shape4p	1	**125**	SE_shape2c	0
19	LSP2p	6	**74**	SE_shape4p	0	**Class VII Bit**		
20	LSP2p	5	**75**	SE_shape5p	7	**126**	LSP2p	2
21	LSP2p	4	**76**	SE_shape5p	6	**127**	LSP2p	1
22	LSP2p	3	**77**	SE_shape5p	5	**128**	LSP2p	0
23	SE_shape3p	6	**78**	SE_shape5p	4	**129**	LSP3p	3
24	SE_shape3p	5	**79**	SE_shape5p	3	**130**	LSP3p	2
25	SE_shape3p	4	**80**	SE_shape5p	2	**131**	LSP3p	1
26	SE_shape3p	3	**81**	SE_shape5p	1	**132**	LSP3p	0
27	SE_shape3p	2	**82**	SE_shape5p	0	**133**	LSP5p	6
28	LSP3p	4	**83**	SE_shape6p	4	**134**	LSP5p	5
29	LSP5p	7	**84**	SE_shape6p	3	**135**	LSP5p	4
30	SE_shape4p	9	**85**	SE_shape6p	2	**136**	LSP5p	3
31	SE_shape5p	8	**86**	SE_shape6p	1	**137**	LSP5p	2
32	SE_shape6p	5	**87**	SE_shape6p	0	**138**	LSP5p	1
33	VUVc	1	**88**	SE_shape4c	8	**139**	LSP5p	0
34	VUVc	0	**89**	SE_shape4c	7	**140**	Pitchp	0
35	LSP4c	0	**90**	SE_shape4c	6	**141**	SE_shape3p	1
36	SE_gainc	4	**91**	SE_shape4c	5	**142**	SE_shape3p	0
37	SE_gainc	3	**92**	SE_shape4c	4	**143**	LSP2c	2
38	SE_gainc	2	**93**	SE_shape4c	3	**144**	LSP2c	1
39	SE_gainc	1	**94**	SE_shape4c	2	**145**	LSP2c	0
40	SE_gainc	0	**95**	SE_shape4c	1	**146**	LSP3c	3
41	LSP1c	4	**96**	SE_shape4c	0	**147**	LSP3c	2
42	LSP1c	3	**97**	SE_shape5c	7	**148**	LSP3c	1
43	LSP1c	2	**98**	SE_shape5c	6	**149**	LSP3c	0
44	LSP1c	1	**99**	SE_shape5c	5	**150**	LSP5c	6
45	LSP1c	0	**100**	SE_shape5c	4	**151**	LSP5c	5
46	Pitchc	6	**101**	SE_shape5c	3	**152**	LSP5c	4
47	Pitchc	5	**102**	SE_shape5c	2	**153**	LSP5c	3
48	Pitchc	4	**103**	SE_shape5c	1	**154**	LSP5c	2
49	Pitchc	3	**104**	SE_shape5c	0	**155**	LSP5c	1
50	Pitchc	2	**105**	SE_shape6c	4	**156**	LSP5c	0
51	Pitchc	1	**106**	SE_shape6c	3	**157**	Pitchc	0
52	LSP2c	6	**107**	SE_shape6c	2	**158**	SE_shape3c	1
53	LSP2c	5	**108**	SE_shape6c	1	**159**	SE_shape3c	0
54	LSP2c	4	**109**	SE_shape6c	0			

Table 1.B.13 — Bit Order for 4 kbit/s (voiced frame – unvoiced frame)

voiced frame – unvoiced frame								
No.	**Item**	**Bit**	**No.**	**Item**	**Bit**	**No.**	**Item**	**Bit**
Class 1bit			**55**	VX_gain2[0]c	2	**Class III Bit**		
0	VUVp	1	**56**	VX_gain2[0]c	1	**110**	SE_shape1p	3
1	VUVp	0	**57**	VX_gain2[0]c	0	**111**	SE_shape1p	2
2	LSP4p	0	**58**	VX_gain2[1]c	2	**112**	SE_shape1p	1
3	SE_gainp	4	**59**	VX_gain2[1]c	1	**113**	SE_shape1p	0

4	SE_gainp	3	**60**	VX_gain2[1]c	0	**Class IV Bit**		
5	SE_gainp	2	**61**	VX_gain2[2]c	2	**114**	SE_shape2p	3
6	SE_gainp	1	**62**	VX_gain2[2]c	1	**115**	SE_shape2p	2
7	SE_gainp	0	**63**	VX_gain2[2]c	0	**116**	SE_shape2p	1
8	LSP1p	4	**64**	VX_gain2[3]c	2	**117**	SE_shape2p	0
9	LSP1p	3	**65**	VX_gain2[3]c	1	**Class V Bit**		
10	LSP1p	2	**Class II Bit**			**118**	VX_shape1[1]c	4
11	LSP1p	1	**66**	SE_shape4p	8	**119**	VX_shape1[1]c	3
12	LSP1p	0	**67**	SE_shape4p	7	**120**	VX_shape1[1]c	2
13	Pitchp	6	**68**	SE_shape4p	6	**121**	VX_shape1[1]c	1
14	Pitchp	5	**69**	SE_shape4p	5	**Class VI Bit**		
15	Pitchp	4	**70**	SE_shape4p	4	**122**	VX_shape1[1]c	0
16	Pitchp	3	**71**	SE_shape4p	3	**123**	VX_shape2[0]c	4
17	Pitchp	2	**72**	SE_shape4p	2	**124**	VX_shape2[0]c	3
18	Pitchp	1	**73**	SE_shape4p	1	**125**	VX_shape2[0]c	2
19	LSP2p	6	**74**	SE_shape4p	0	**Class VII Bit**		
20	LSP2p	5	**75**	SE_shape5p	7	**126**	LSP2p	2
21	LSP2p	4	**76**	SE_shape5p	6	**127**	LSP2p	1
22	LSP2p	3	**77**	SE_shape5p	5	**128**	LSP2p	0
23	SE_shape3p	6	**78**	SE_shape5p	4	**129**	LSP3p	3
24	SE_shape3p	5	**79**	SE_shape5p	3	**130**	LSP3p	2
25	SE_shape3p	4	**80**	SE_shape5p	2	**131**	LSP3p	1
26	SE_shape3p	3	**81**	SE_shape5p	1	**132**	LSP3p	0
27	SE_shape3p	2	**82**	SE_shape5p	0	**133**	LSP5p	6
28	LSP3p	4	**83**	SE_shape6p	4	**134**	LSP5p	5
29	LSP5p	7	**84**	SE_shape6p	3	**135**	LSP5p	4
30	SE_shape4p	9	**85**	SE_shape6p	2	**136**	LSP5p	3
31	SE_shape5p	8	**86**	SE_shape6p	1	**137**	LSP5p	2
32	SE_shape6p	5	**87**	SE_shape6p	0	**138**	LSP5p	1
33	VUVc	1	**88**	VX_gain2[3]c	0	**139**	LSP5p	0
34	VUVc	0	**89**	LSP2c	2	**140**	Pitchp	0
35	LSP4c	0	**90**	LSP2c	1	**141**	SE_shape3p	1
36	VX_gain1[0]c	3	**91**	LSP2c	0	**142**	SE_shape3p	0
37	VX_gain1[0]c	2	**92**	LSP3c	3	**143**	VX_shape2[0]c	1
38	VX_gain1[0]c	1	**93**	LSP3c	2	**144**	VX_shape2[0]c	0
39	VX_gain1[0]c	0	**94**	LSP3c	1	**145**	VX_shape2[1]c	4
40	VX_gain1[1]c	3	**95**	LSP3c	0	**146**	VX_shape2[1]c	3
41	VX_gain1[1]c	2	**96**	LSP5c	6	**147**	VX_shape2[1]c	2
42	VX_gain1[1]c	1	**97**	LSP5c	5	**148**	VX_shape2[1]c	1
43	VX_gain1[1]c	0	**98**	LSP5c	4	**149**	VX_shape2[1]c	0
44	LSP1c	4	**99**	LSP5c	3	**150**	VX_shape2[2]c	4
45	LSP1c	3	**100**	LSP5c	2	**151**	VX_shape2[2]c	3
46	LSP1c	2	**101**	LSP5c	1	**152**	VX_shape2[2]c	2
47	LSP1c	1	**102**	LSP5c	0	**153**	VX_shape2[2]c	1
48	LSP1c	0	**103**	VX_shape1[0]c	5	**154**	VX_shape2[2]c	0
49	LSP2c	6	**104**	VX_shape1[0]c	4	**155**	VX_shape2[3]c	4
50	LSP2c	5	**105**	VX_shape1[0]c	3	**156**	VX_shape2[3]c	3
51	LSP2c	4	**106**	VX_shape1[0]c	2	**157**	VX_shape2[3]c	2
52	LSP2c	3	**107**	VX_shape1[0]c	1	**158**	VX_shape2[3]c	1

53	LSP3c	4	**108**	VX_shape1[0]c	0	**159**	VX_shape2[3]c	0
54	LSP5c	7	**109**	VX_shape1[1]c	5			

Table 1.B.14 — Bit Order for 4 kbit/s (unvoiced frame – voiced frame)

unvoiced frame – voiced frame								
No.	**Item**	**Bit**	**No.**	**Item**	**Bit**	**No.**	**Item**	**Bit**
Class 1bit			**55**	LSP2c	3	**Class III Bit**		
0	VUVp	1	**56**	SE_shape3c	6	**110**	VX_shape1[1]p	4
1	VUVp	0	**57**	SE_shape3c	5	**111**	VX_shape1[1]p	3
2	LSP4p	0	**58**	SE_shape3c	4	**112**	VX_shape1[1]p	2
3	VX_gain1[0]p	3	**59**	SE_shape3c	3	**113**	VX_shape1[1]p	1
4	VX_gain1[0]p	2	**60**	SE_shape3c	2	**Class IV Bit**		
5	VX_gain1[0]p	1	**61**	LSP3c	4	**114**	VX_shape1[1]p	0
6	VX_gain1[0]p	0	**62**	LSP5c	7	**115**	VX_shape2[0]p	4
7	VX_gain1[1]p	3	**63**	SE_shape4c	9	**116**	VX_shape2[0]p	3
8	VX_gain1[1]p	2	**64**	SE_shape5c	8	**117**	VX_shape2[0]p	2
9	VX_gain1[1]p	1	**65**	SE_shape6c	5	**Class V Bit**		
10	VX_gain1[1]p	0	**Class II Bit**			**118**	SE_shape1c	3
11	LSP1p	4	**66**	VX_gain2[3]p	0	**119**	SE_shape1c	2
12	LSP1p	3	**67**	LSP2p	2	**120**	SE_shape1c	1
13	LSP1p	2	**68**	LSP2p	1	**121**	SE_shape1c	0
14	LSP1p	1	**69**	LSP2p	0	**Class VI Bit**		
15	LSP1p	0	**70**	LSP3p	3	**122**	SE_shape2c	3
16	LSP2p	6	**71**	LSP3p	2	**123**	SE_shape2c	2
17	LSP2p	5	**72**	LSP3p	1	**124**	SE_shape2c	1
18	LSP2p	4	**73**	LSP3p	0	**125**	SE_shape2c	0
19	LSP2p	3	**74**	LSP5p	6	**Class VII Bit**		
20	LSP3p	4	**75**	LSP5p	5	**126**	VX_shape2[0]p	1
21	LSP5p	7	**76**	LSP5p	4	**127**	VX_shape2[0]p	0
22	VX_gain2[0]p	2	**77**	LSP5p	3	**128**	VX_shape2[1]p	4
23	VX_gain2[0]p	1	**78**	LSP5p	2	**129**	VX_shape2[1]p	3
24	VX_gain2[0]p	0	**79**	LSP5p	1	**130**	VX_shape2[1]p	2
25	VX_gain2[1]p	2	**80**	LSP5p	0	**131**	VX_shape2[1]p	1
26	VX_gain2[1]p	1	**81**	VX_shape1[0]p	5	**132**	VX_shape2[1]p	0
27	VX_gain2[1]p	0	**82**	VX_shape1[0]p	4	**133**	VX_shape2[2]p	4
28	VX_gain2[2]p	2	**83**	VX_shape1[0]p	3	**134**	VX_shape2[2]p	3
29	VX_gain2[2]p	1	**84**	VX_shape1[0]p	2	**135**	VX_shape2[2]p	2
30	VX_gain2[2]p	0	**85**	VX_shape1[0]p	1	**136**	VX_shape2[2]p	1
31	VX_gain2[3]p	2	**86**	VX_shape1[0]p	0	**137**	VX_shape2[2]p	0
32	VX_gain2[3]p	1	**87**	VX_shape1[1]p	5	**138**	VX_shape2[3]p	4
33	VUVc	1	**88**	SE_shape4c	8	**139**	VX_shape2[3]p	3
34	VUVc	0	**89**	SE_shape4c	7	**140**	VX_shape2[3]p	2
35	LSP4c	0	**90**	SE_shape4c	6	**141**	VX_shape2[3]p	1
36	SE_gainc	4	**91**	SE_shape4c	5	**142**	VX_shape2[3]p	0
37	SE_gainc	3	**92**	SE_shape4c	4	**143**	LSP2c	2
38	SE_gainc	2	**93**	SE_shape4c	3	**144**	LSP2c	1
39	SE_gainc	1	**94**	SE_shape4c	2	**145**	LSP2c	0

40	SE_gainc	0	**95**	SE_shape4c	1	**146**	LSP3c	3
41	LSP1c	4	**96**	SE_shape4c	0	**147**	LSP3c	2
42	LSP1c	3	**97**	SE_shape5c	7	**148**	LSP3c	1
43	LSP1c	2	**98**	SE_shape5c	6	**149**	LSP3c	0
44	LSP1c	1	**99**	SE_shape5c	5	**150**	LSP5c	6
45	LSP1c	0	**100**	SE_shape5c	4	**151**	LSP5c	5
46	Pitchc	6	**101**	SE_shape5c	3	**152**	LSP5c	4
47	Pitchc	5	**102**	SE_shape5c	2	**153**	LSP5c	3
48	Pitchc	4	**103**	SE_shape5c	1	**154**	LSP5c	2
49	Pitchc	3	**104**	SE_shape5c	0	**155**	LSP5c	1
50	Pitchc	2	**105**	SE_shape6c	4	**156**	LSP5c	0
51	Pitchc	1	**106**	SE_shape6c	3	**157**	Pitchc	0
52	LSP2c	6	**107**	SE_shape6c	2	**158**	SE_shape3c	1
53	LSP2c	5	**108**	SE_shape6c	1	**159**	SE_shape3c	0
54	LSP2c	4	**109**	SE_shape6c	0			

Table 1.B.15 — Bit Order for 4 kbit/s (unvoiced frame – unvoiced frame)

unvoiced frame – unvoiced frame								
No.	**Item**	**Bit**	**No.**	**Item**	**Bit**	**No.**	**Item**	**Bit**
Class 1bit			**55**	VX_gain2[0]c	2	**Class III Bit**		
0	VUVp	1	**56**	VX_gain2[0]c	1	**110**	VX_shape1[1]p	4
1	VUVp	0	**57**	VX_gain2[0]c	0	**111**	VX_shape1[1]p	3
2	LSP4p	0	**58**	VX_gain2[1]c	2	**112**	VX_shape1[1]p	2
3	VX_gain1[0]p	3	**59**	VX_gain2[1]c	1	**113**	VX_shape1[1]p	1
4	VX_gain1[0]p	2	**60**	VX_gain2[1]c	0	**Class IV Bit**		
5	VX_gain1[0]p	1	**61**	VX_gain2[2]c	2	**114**	VX_shape1[1]p	0
6	VX_gain1[0]p	0	**62**	VX_gain2[2]c	1	**115**	VX_shape2[0]p	4
7	VX_gain1[1]p	3	**63**	VX_gain2[2]c	0	**116**	VX_shape2[0]p	3
8	VX_gain1[1]p	2	**64**	VX_gain2[3]c	2	**117**	VX_shape2[0]p	2
9	VX_gain1[1]p	1	**65**	VX_gain2[3]c	1	**Class V Bit**		
10	VX_gain1[1]p	0	**Class II Bit**			**118**	VX_shape1[1]c	4
11	LSP1p	4	**66**	VX_gain2[3]p	0	**119**	VX_shape1[1]c	3
12	LSP1p	3	**67**	LSP2p	2	**120**	VX_shape1[1]c	2
13	LSP1p	2	**68**	LSP2p	1	**121**	VX_shape1[1]c	1
14	LSP1p	1	**69**	LSP2p	0	**Class VI Bit**		
15	LSP1p	0	**70**	LSP3p	3	**122**	VX_shape1[1]c	0
16	LSP2p	6	**71**	LSP3p	2	**123**	VX_shape2[0]c	4
17	LSP2p	5	**72**	LSP3p	1	**124**	VX_shape2[0]c	3
18	LSP2p	4	**73**	LSP3p	0	**125**	VX_shape2[0]c	2
19	LSP2p	3	**74**	LSP5p	6	**Class VII Bit**		
20	LSP3p	4	**75**	LSP5p	5	**126**	VX_shape2[0]p	1
21	LSP5p	7	**76**	LSP5p	4	**127**	VX_shape2[0]p	0
22	VX_gain2[0]p	2	**77**	LSP5p	3	**128**	VX_shape2[1]p	4
23	VX_gain2[0]p	1	**78**	LSP5p	2	**129**	VX_shape2[1]p	3
24	VX_gain2[0]p	0	**79**	LSP5p	1	**130**	VX_shape2[1]p	2
25	VX_gain2[1]p	2	**80**	LSP5p	0	**131**	VX_shape2[1]p	1
26	VX_gain2[1]p	1	**81**	VX_shape1[0]p	5	**132**	VX_shape2[1]p	0

27	VX_gain2[1]p	0	**82**	VX_shape1[0]p	4	**133**	VX_shape2[2]p	4
28	VX_gain2[2]p	2	**83**	VX_shape1[0]p	3	**134**	VX_shape2[2]p	3
29	VX_gain2[2]p	1	**84**	VX_shape1[0]p	2	**135**	VX_shape2[2]p	2
30	VX_gain2[2]p	0	**85**	VX_shape1[0]p	1	**136**	VX_shape2[2]p	1
31	VX_gain2[3]p	2	**86**	VX_shape1[0]p	0	**137**	VX_shape2[2]p	0
32	VX_gain2[3]p	1	**87**	VX_shape1[1]p	5	**138**	VX_shape2[3]p	4
33	VUVc	1	**88**	VX_gain2[3]c	0	**139**	VX_shape2[3]p	3
34	VUVc	0	**89**	LSP2c	2	**140**	VX_shape2[3]p	2
35	LSP4c	0	**90**	LSP2c	1	**141**	VX_shape2[3]p	1
36	VX_gain1[0]c	3	**91**	LSP2c	0	**142**	VX_shape2[3]p	0
37	VX_gain1[0]c	2	**92**	LSP3c	3	**143**	VX_shape2[0]c	1
38	VX_gain1[0]c	1	**93**	LSP3c	2	**144**	VX_shape2[0]c	0
39	VX_gain1[0]c	0	**94**	LSP3c	1	**145**	VX_shape2[1]c	4
40	VX_gain1[1]c	3	**95**	LSP3c	0	**146**	VX_shape2[1]c	3
41	VX_gain1[1]c	2	**96**	LSP5c	6	**147**	VX_shape2[1]c	2
42	VX_gain1[1]c	1	**97**	LSP5c	5	**148**	VX_shape2[1]c	1
43	VX_gain1[1]c	0	**98**	LSP5c	4	**149**	VX_shape2[1]c	0
44	LSP1c	4	**99**	LSP5c	3	**150**	VX_shape2[2]c	4
45	LSP1c	3	**100**	LSP5c	2	**151**	VX_shape2[2]c	3
46	LSP1c	2	**101**	LSP5c	1	**152**	VX_shape2[2]c	2
47	LSP1c	1	**102**	LSP5c	0	**153**	VX_shape2[2]c	1
48	LSP1c	0	**103**	VX_shape1[0]c	5	**154**	VX_shape2[2]c	0
49	LSP2c	6	**104**	VX_shape1[0]c	4	**155**	VX_shape2[3]c	4
50	LSP2c	5	**105**	VX_shape1[0]c	3	**156**	VX_shape2[3]c	3
51	LSP2c	4	**106**	VX_shape1[0]c	2	**157**	VX_shape2[3]c	2
52	LSP2c	3	**107**	VX_shape1[0]c	1	**158**	VX_shape2[3]c	1
53	LSP3c	4	**108**	VX_shape1[0]c	0	**159**	VX_shape2[3]c	0
54	LSP5c	7	**109**	VX_shape1[1]c	5			

1.B.3.3 EP tool setting

1.B.3.3.1 Bit assignment

The Table 1.B.16 below shows an example bit assignment for the use of the EP tool. In this table, bit assignments for both of the 2 kbit/s and 4 kbit/s source coder are described.

Table 1.B.16 — The bit assignment for the use of the EP tool

	2 kbit/s Source Coder	4 kbit/s Source Coder
Class I		
Source coder bits	44	66
CRC parity	6	6
Code Rate	8/16	8/16
Class I total	100	144
Class II		
Source coder bits	4	44
CRC parity	1	6
Code Rate	8/8	8/8
Class II total	5	50
Class III		
Source coder bits	4	4
CRC parity	1	1
Code Rate	8/8	8/8
Class III total	5	5
Class IV		
Source coder bits	4	4
CRC parity	1	1
Code Rate	8/8	8/8
Class IV total	5	5
Class V		
Source coder bits	4	4
CRC parity	1	1
Code Rate	8/8	8/8
Class V total	5	5
Class VI		
Source coder bits	20	4
CRC parity	0	1
Code Rate	8/8	8/8
Class VI total	20	5
Class VII		
Source coder bits		34
CRC parity		0
Code Rate		8/8
Class VII total		34
Total Bit of All Classes	140	248
Bitrate	3.5 kbit/s	6.2 kbit/s

Class I:

CRC covers all the Class I bits, and Class I bits including CRC are protected by convolutional coding.

Class II-V(2 kbit/s), II-VI(4 kbit/s):

At least one CRC bits cover the source coder bits of these classes.

Class VI(2 kbit/s), VII(4 kbit/s) :

The source coder bits are not checked by CRC nor protected by any error correction scheme.

1.B.3.4 Error concealment

When CRC error is detected, error concealment processing (bad frame masking) is carried out. An example of concealment method is described below.

A frame masking state of the current frame is updated based on the decoded CRC result of Class I. The state transition diagram is shown in Figure 1.B.2. The initial state is state = 0. The arrow with a letter “1” denotes the transition for a bad frame, and that with a letter “0” a good frame.

1.B.3.4.1 Parameter replacemet

According to the state value, the following parameter replacement is done. In error free condition, state value becomes 0, and received source coder bits are used without any concealment processing.

1.B.3.4.1.1 LSP parameters

At state=1..6, LSP parameters are replaced with those of previous ones.

When state=7, If LSP4=0 (LSP quantization mode without inter-frame prediction), then LSP parameters are calculated from all LSP indices received in the current frame. If LSP4=1 (LSP quantization mode with inter-frame coding), then LSP parameters are calculated with the following method.

In this mode, LSP parameters from LSP1 index are interpolated with the previous LSPs.

$$LSP_{base}(n) = p \cdot LSP_{prev}(n) + (1-p)LSP_{1st}(n) \qquad \text{for n=1..10} \quad (1)$$

$LSP_{base}(n)$ is LSP parameters of the base layer, $LSP_{prev}(n)$ is the previous LSPs, $LSP_{1st}(n)$ is the decoded LSPs from the current LSP1 index, and p is the factor of interpolation. p is changed according to the number of previous CRC error frames of Class I bits as shown in Table 1.B.17. LSP indices LSP2, LSP3 and LSP5 are not used and $LSP_{base}(n)$ is used as current LSP parameters.

Table 1.B.17 — p factor

frame	p
0	0.7
1	0.6
2	0.5
3	0.4
4	0.3
5	0.2
6	0.1
≥7	0.0

1.B.3.4.1.2 Mute variable

According to the “state” value, a variable “mute” is set to control output level of speech.
The “mute” value below is used.
In state = 7, the average of 1.0 and “mute” value of the previous frame (= 0.5 (1.0 + previous “mute value”)) is used, but when this value is more than 0.8, “mute” value is replaced with 0.8.

Table 1.B.18 — mute value

State	mute
0	1.000
1	0.800
2	0.700
3	0.500
4	0.250
5	0.125
6	0.000
7	Average/0.800

1.B.3.4.1.3 Replacement and gain control of "voiced" parameters

In state=1..6, spectrum parameter SE_shape1, SE_shape2, spectrum gain parameter SE_gain, spectrum parameter for 4 kbit/s codec SE_shape3 .. SE_shape6 are replaced with corresponding parameters of the previous frame. Also, to control volume of output speech, harmonic magnitude parameters of LPC residual signal "$Am[0\ldots127]$" is gain controlled as shown in Eq.(1). In the equation, $Am_{(org)}[i]$ is computed from the received spectrum parameters from the latest error free frame.

$$Am[i] = mute * Am_{(org)}[i] \quad \text{for i=0..127} \qquad (1)$$

If previous frame is unvoiced and current state is state=7, Eq.(1) is replaced with Eq.(2).

$$Am[i] = 0.6 * mute * Am_{(org)}[i] \quad \text{for i=0..127} \qquad (2)$$

As described before, SE_shape1 and SE_shape2 are individually protected by 1 bit CRC. In state=0 or 7, when CRC errors of these classes are detected at the same time, the quantized harmonic magnitudes with fixed dimension $Am_{qnt}[1..44]$ are gain suppressed as shown in Eq.(3).

$$Am_{qnt}[i] = s[i] * Am_{qnt(org)}[i] \quad \text{for i=1..44} \qquad (3)$$

$s[i]$ is the factor for the gain suppression.

Table 1.B.19 — factor for gain suppression 's[0..44]'

i	1	2	3	4	5	6	7..44
$s[i]$	0.10	0.25	0.40	0.55	0.70	0.85	1.00

At 4 kbit/s, SE_shape4, SE_shape5, and SE_shape6 are checked by CRC as Class II bits. When CRC error is detected, the spectrum parameter of the enhancement layer is not used.

1.B.3.4.1.4 Replacement and gain control of "unvoiced" parameters

In state=1..6, stochastic codebook gain parameter VX_gain1[0], VX_gain1[1] are replaced with the VX_gain1[1] from the latest error free frame. Also stochastic codebook gain parameter for 4 kbit/s codec VX_gain2[0]..VX_gain2[3] are replaced with the VX_gain2[3] from the latest error free frame.
Stochastic codebook shape parameter VX_shape1[0], VX_shape1[1],and stochastic codebook shape parameter for 4 kbit/s codec are generated from randomly generated index values.
Also, to control volume of output speech, LPC residual signal $res[0\ldots159]$ is gain controlled as shown in Eq.(4). In the equation, $res_{(org)}[i]$ is computed from stochastic codebook parameters.

$$res[i] = mute * res_{(org)}[i] \quad (0 \le i \le 159) \qquad (4)$$

1.B.3.4.1.5 Frame Masking State Transitions

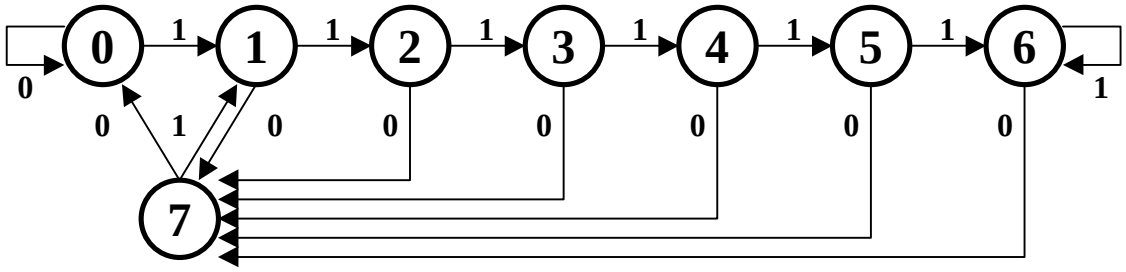


Figure 1.B.2 — Frame Masking State Transitions

Annex 1.C
(informative)

Patent statements

The International Organization for Standardization and the International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this part of ISO/IEC 14496 may involve the use of patents.

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights.

The holders of these patent rights have assured the ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statements of the holders of these patents right are registered with ISO and IEC. Information may be obtained from the companies listed in the Table below.

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 14496 may be the subject of patent rights other than those identified in this annex. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Alcatel
AT&T
BBC
Bosch
British Telecommunications
Canon
CCETT
Coding Technologies
Columbia Innovation Enterprise
Columbia University
Creative
CSELT
DemoGraFX
DirecTV
Dolby
EPFL
ETRI
France Telecom
Fraunhofer
Fujitsu
GC Technology Corporation
Hitachi
Hyundai
IBM
Institut fuer Rundfunktechnik
JVC
KPN
Matsushita Electric Industrial Co., Ltd.
Microsoft
Mitsubishi
NEC
NHK
Nokia
NTT

NTT Mobile Communication Networks
OKI
Philips
Rockwell
Samsung
Sarnoff
Scientific Atlanta
Sharp
Siemens
Sony
Telenor
Thomson

Content for Subpart 2

Subpart 2: Speech coding - HVXC

2.1 Scope

MPEG-4 parametric speech coding uses Harmonic Vector eXcitation Coding (HVXC) algorithm, where harmonic coding of LPC residual signals for voiced segments and Vector eXcitation Coding (VXC) for unvoiced segments are employed. HVXC allows coding of speech signals at 2.0 kbps and 4.0 kbps with a scalable scheme, where 2.0 kbps decoding is possible not only using the 2.0 kbps bitstream but also using a 4.0 kbps bitstream. HVXC also provides variable bit rate coding where a typical average bit-rate is around 1.2-1.7 kbit/s. Independent change of speed and pitch during decoding is possible, which is a powerful functionality for fast data base search. The frame length is 20 ms, and one of four different algorithmic delays, 33.5 ms, 36ms, 53.5 ms, 56 ms can be selected.

Furthermore as an extension of HVXC, ER_HVXC object type offers error resilient syntax and the 4.0 kbit/s variable bitrate mode.

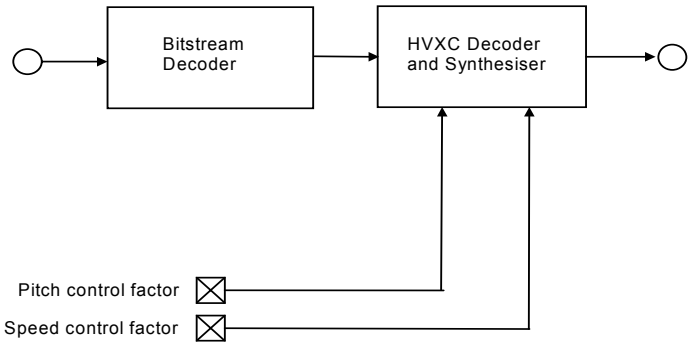


Figure 2.1 — Block diagram of the parametric speech decoder

2.2 Definitions

Definitions can be found in subpart 1, subclause 1.3.

2.3 Bitstream syntax

An MPEG-4 Natural Audio Object HVXC/ER_HVXC object type is transmitted in one or two Elementary Streams: The base layer stream and an optional enhancement layer stream.

When the HVXC tool is used with an error protection tool, such as an MPEG-4 EP tool, the bit order arranged in accordance with the error sensitivity should be used. The HVXC with the error resilient syntax and 4.0 kbit/s variable bitrate mode described in subclauses 2.3.3 and 2.3.4 are called ER_HVXC.

The bitstream syntax is described in pseudo-C code.

2.3.1 Decoder configuration (HvxcSpecificConfig)

The decoder configuration information for HVXC object type is transmitted in the DecoderConfigDescriptor() of the base layer and the optional enhancement layer Elementary Stream (see subpart 1, subclause 1.6).

The following HvxcSpecificConfig() is required:

```
HvxcSpecificConfig ( ) {
    isBaseLayer                 1   uimsbf
    if (isBaseLayer) {
        HVXCconfig()
    }
}
```

HVXC object type provides unscalable modes and a 2 kbit/s base layer plus a 2 kbit/s enhancement layer scalable mode. In this scalable mode the basic layer configuration must be as follows:

```
HVXCvarMode = 0      HVXC fixed bit rate
HVXCrateMode = 0     HVXC 2kbps
isBaseLayer=1        base layer
```

Table 2.1 — Syntax of HVXCconfig()

Syntax	No. of bits	Mnemonic
HVXCconfig()		
{		
HVXCvarMode;	**1**	**uimsbf**
HVXCrateMode;	**2**	**uimsbf**
extensionFlag;	**1**	**uimsbf**
if (extensionFlag) {		
< to be defined in MPEG-4 Version 2 >		
}		
}		

Table 2.2 — HVXCvarMode

HVXCvarMode	Description
0	HVXC fixed bit rate
1	HVXC variable bit rate

Table 2.3 — HVXCrateMode

HVXCrateMode	HVXCrate	Description
0	2000	HVXC 2 kbit/s
1	4000	HVXC 4 kbit/s
2	3700	HVXC 3.7 kbit/s
3 (reserved)		

Table 2.4 — HVXC constants

NUM_SUBF1	NUM_SUBF2
2	4

2.3.2 Bitstream frame (alPduPayload)

The dynamic data for HVXC object type is transmitted as AL-PDU payload in the base layer and the optional enhancement layer Elementary Stream (see subpart 1, subclause 1.6).

HVXC Base Layer -- Access Unit payload

```
alPduPayload {
   HVXCframe();
}
```

HVXC Enhancement Layer -- Access Unit payload

To parse and decode the HVXC enhancement layer, information decoded from the HVXC base layer is required.

```
alPduPayload {
   HVXCenhaFrame();
}
```

Table 2.5 — Syntax of HVXCframe()

Syntax	No. of bits	Mnemonic

HVXCframe()
{
  if (HVXCvarMode == 0) {
  HVXCfixframe(HVXCrate);
  }
  else {
  HVXCvarframe();
  }
}		

2.3.2.1 HVXC bitstream frame

Table 2.6 — Syntax of HVXCfixframe(rate)

Syntax	No. of bits	Mnemonic

HVXCfixframe(rate)
{
  if (rate >= 2000) {
  idLsp1();
  idVUV();
  idExc1();
  }
  if (rate >= 3700) {
  idLsp2();
  idExc2(rate);
  }
}		

Table 2.7 — Syntax of HVXCenhaFrame()

Syntax	No. of bits	Mnemonic

HVXCenhaFrame()
{
  idLsp2();
  idExc2(4000);
}		

Table 2.8 — Syntax of idLsp1()

Syntax	No. of bits	Mnemonic
idLsp1()		
{		
LSP1;	**5**	**uimsbf**
LSP2;	**7**	**uimsbf**
LSP3;	**5**	**uimsbf**
LSP4;	**1**	**uimsbf**
}		

Table 2.9 — Syntax of idLsp2()

Syntax	No. of bits	Mnemonic
idLsp2()		
{		
LSP5;	**8**	**uimsbf**
}		

Table 2.10 — Syntax of idVUV()

Syntax	No. of bits	Mnemonic
idVUV()		
{		
VUV;	**2**	**uimsbf**
}		

Table 2.11 — VUV (for fixed bit rate mode)

VUV	Description
0	Unvoiced Speech
1	Mixed Voiced Speech-1
2	Mixed Voiced Speech-2
3	Voiced Speech

Table 2.12 — Syntax of idExc1()

Syntax	No. of bits	Mnemonic
idExc1()		
{		
if (VUV != 0) {		
Pitch;	**7**	**uimsbf**
SE_shape1;	**4**	**uimsbf**
SE_shape2;	**4**	**uimsbf**
SE_gain;	**5**	**uimsbf**
}		
else{		
for (sf_num = 0; sf_num < NUM_SUBF1; sf_num++) {		
VX_shape1 [sf_num];	**6**	**uimsbf**
VX_gain1 [sf_num];	**4**	**uimsbf**
}		
}		

```
}
```

Table 2.13 — Syntax of idExc2(rate)

Syntax	No. of bits	Mnemonic
idExc2(rate)		
{		
if (VUV != 0) {		
SE_shape3;	**7**	**uimsbf**
SE_shape4;	**10**	**uimsbf**
SE_shape5;	**9**	**uimsbf**
if (rate >= 4000) {		
SE_shape6;	**6**	**uimsbf**
}		
}		
else {		
for (sf_num = 0; sf_num < NUM_SUBF2-1; sf_num++) {		
VX_shape2[sf_num];	**5**	**uimsbf**
VX_gain2[sf_num];	**3**	**uimsbf**
}		
if (rate >= 4000) {		
VX_shape2[3];	**5**	**uimsbf**
VX_gain2[3];	**3**	**uimsbf**
}		
}		
}		

idLsp1(), idExc1(), idVUV() are treated as base layer in case of scalable mode.

idLsp2(), idExc2() are treated as enhancement layer in case of scalable mode.

Table 2.14 — Syntax of HVXCvarframe()

Syntax	No. of bits	Mnemonic
HVXCvarframe()		
{		
idvarVUV();		
idvarLsp1();		
idvarExc1();		
}		

Table 2.15 — Syntax of idvarVUV()

Syntax	No. of bits	Mnemonic
idvarVUV()		
{		
VUV;	**2**	**uimsbf**
}		

Table 2.16 — VUV (for variable bit rate mode)

VUV	Description
0	Unvoiced Speech
1	Background Noise

2	Mixed Voiced Speech
3	Voiced Speech

Table 2.17 — Syntax of idvarLsp1()

Syntax	No. of bits	Mnemonic
idvarLsp1()		
{		
if (VUV != 1) {		
LSP1;	**5**	**uimsbf**
LSP2;	**7**	**uimsbf**
LSP3;	**5**	**uimsbf**
LSP4;	**1**	**uimsbf**
}		
}		

Table 2.18 — Syntax of idvarExc1()

Syntax	No. of bits	Mnemonic
idvarExc1()		
{		
if (VUV != 1) {		
if (VUV != 0) {		
Pitch;	**7**	**uimsbf**
SE_Shape1;	**4**	**uimsbf**
SE_Shape2;	**4**	**uimsbf**
SE_Gain;	**5**	**uimsbf**
}		
else {		
for (sf_num = 0;sf_num < NUM_SUBF1; sf_num++) {		
VX_gain1[sf_num];	**4**	**uimsbf**
}		
}		
}		
}		

2.3.3 Decoder configuration (ErrorResilientHvxcSpecificConfig)

The decoder configuration information for ER_HVXC object type is transmitted in the DecoderConfigDescriptor() of the base layer and the optional enhancement layer Elementary Stream.

The following ErrorResilientHvxcSpecificConfig() is required:

```
ErrorResilientHvxcSpecificConfig() {
    isBaseLayer                 1   uimsbf
    if  (isBaseLayer) {
        ErHVXCconfig();
    }
}
```

ER_HVXC object type provides unscalable modes and scalable modes. In the scalable modes the base layer configuration must be as follows:

HVXCrateMode = 0 ER_HVXC 2 kbit/s
isBaseLayer = 1 base layer

Table 2.19 — Syntax of ErHvxcConfig()

Syntax	No. of bits	Mnemonic
ErHVXCconfig()		
{		
HVXCvarMode;	**1**	**uimsbf**
HVXCrateMode;	**2**	**uimsbf**
extensionFlag;	**1**	**uimsbf**
if (extensionFlag) {		
var_ScalableFlag;	**1**	**uimsbf**
}		
}		

Table 2.20 — HVXCvarMode

HVXCvarMode	Description
0	ER_HVXC fixed bitrate
1	ER_HVXC variable bitrate

Table 2.21 — HVXCrateMode

HVXCrateMode	HVXCrate	Description
0	2000	ER_HVXC 2 kbit/s
1	4000	ER_HVXC 4 kbit/s
2	3700	ER_HVXC 3.7 kbit/s
3 (reserved)		

Table 2.22 — var_ScalableFlag

var_ScalableFlag	Description
0	ER_HVXC variable rate non-scalable mode
1	ER_HVXC variable rate scalable mode

2.3.4 Bitstream frame (alPduPayload)

The dynamic data for ER_HVXC object type is transmitted as AL-PDU payload in the base layer and the optional enhancement layer Elementary Stream.

ER_HVXC Base Layer -- Access Unit payload

```
alPduPayload {
    ErHVXCframe();
}
```

ER_HVXC Enhancement Layer -- Access Unit payload

To parse and decode the ER_HVXC enhancement layer, information decoded from the ER_HVXC base layer is required.

```
alPduPayload {
    ErHVXCenhaFrame();
}
```

Table 2.23 — Syntax of ErHVXCframe()

Syntax	No. of bits	Mnemonic

ErHVXCframe()
{
if (HVXCvarMode == 0) {
ErHVXCfixframe(HVXCrate);
}
else {
ErHVXCvarframe(HVXCrate);
}
}		

Table 2.24 — Syntax of ErHVXCenhaframe()

Syntax	No. of bits	Mnemonic

ErHVXCenhaframe()
{
if (HVXCvarMode == 0) {
ErHVXCenh_fixframe();
}
else {
ErHVXCenh_varframe();
}
}		

2.3.4.1 Bitstream syntax of the fixed rate mode

Table 2.25 — Syntax of ErHVXCfixframe()

Syntax	No. of bits	Mnemonic

ErHVXCfixframe(rate)
{
if (rate == 2000) {
2k_ESC0();
2k_ESC1();
2k_ESC2();
2k_ESC3();
}
else if (rate >= 3700) {
4k_ESC0(rate);
4k_ESC1(rate);
4k_ESC2();
4k_ESC3();
4k_ESC4(rate);
}
}		

Table 2.26 — Syntax of 2k_ESC0()

Syntax	No. of bits	Mnemonic
2k_ESC0() { **VUV , 1-0;**	**2**	**uimsbf**

if (VUV != 0) {		
LSP4, 0;	**1**	**uimsbf**
SE_gain, 4-0;	**5**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
Pitch, 6-1;	**6**	**uimsbf**
LSP2, 6;	**1**	**uimsbf**
LSP3, 4;	**1**	**uimsbf**
LSP2, 5;	**1**	**uimsbf**
}		
else {		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
LSP3, 4-3;	**2**	**uimsbf**
}		
}		

Table 2.27 — Syntax of 2k_ESC1()

Syntax	No. of bits	Mnemonic
2k_ESC1()		
{		
if (VUV != 0) {		
SE_shape1, 3-0;	**4**	**uimsbf**
}		
else {		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2;	**1**	**uimsbf**
}		
}		

Table 2.28 — Syntax of 2k_ESC2()

Syntax	No. of bits	Mnemonic
2k_ESC2()		
{		
if (VUV != 0) {		
SE_shape2, 3-0;	**4**	**uimsbf**
}		
else {		
LSP3, 1-0;	**2**	**uimsbf**
VX_shape1[0], 5-4;	**2**	**uimsbf**
}		
}		

Table 2.29 — Syntax of 2k_ESC3()

Syntax	No. of bits	Mnemonic
2k_ESC3()		
{		
if (VUV != 0) {		
LSP2, 4-0;	**5**	**uimsbf**

LSP3, 3-0;	**4**	**uimsbf**
Pitch, 0;	**1**	**uimsbf**
}		
else {		
VX_shape1[0], 3-0;	**4**	**uimsbf**
VX_shape1[1], 5-0;	**6**	**uimsbf**
}		
}		

Table 2.30 — Syntax of 4k_ESC0()

Syntax	No. of bits	Mnemonic
4k_ESC0()		
{		
VUV , 1-0;	**2**	**uimsbf**
if (VUV != 0) {		
LSP4, 0;	**1**	**uimsbf**
SE_gain, 4-0;	**5**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
Pitch, 6-1;	**6**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
SE_shape3, 6-2;	**5**	**uimsbf**
LSP3, 4;	**1**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
SE_shape4, 9;	**1**	**uimsbf**
SE_shape5, 8;	**1**	**uimsbf**
if (rate >= 4000) {		
SE_shape6, 5;	**1**	**uimsbf**
}		
}		
else {		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
LSP3, 4;	**1**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
VX_gain2[0], 2-0;	**3**	**uimsbf**
VX_gain2[1], 2-0;	**3**	**uimsbf**
VX_gain2[2], 2-0;	**3**	**uimsbf**
if (rate >= 4000) {		
VX_gain2[3], 2-1;	**2**	**uimsbf**
}		
}		
}		

Table 2.31 — Syntax of 4k_ESC1()

Syntax	No. of bits	Mnemonic
4k_ESC1(rate)		
{		
if (VUV != 0) {		
SE_shape4, 8-0;	**9**	**uimsbf**
SE_shape5, 7-0;	**8**	**uimsbf**
if (rate >= 4000) {		
SE_shape6, 4-0;	**5**	**uimsbf**

}
}
else {
if (rate >= 4000) {
VX_gain2[3], 0;
}
LSP2, 2-0;
LSP3, 3-0;
LSP5, 6-0;
VX_shape1[0], 5-0;
VX_shape1[1], 5;
}
}
	1
	
	3
	4
	7
	6
	1
		uimsbf
		
		uimsbf
		uimsbf
		uimsbf
		uimsbf
		uimsbf

Table 2.32 — Syntax of 4k_ESC2()

Syntax	No. of bits	Mnemonic

4k_ESC2()
{
if (VUV != 0) {
SE_shape1, 3-0;
}
else {
VX_shape1[1], 4-1;
}
}
	4
	
	4
		uimsbf
		
		uimsbf

Table 2.33 — Syntax of 4k_ESC3()

Syntax	No. of bits	Mnemonic

4k_ESC3()
{
if (VUV != 0) {
SE_shape2, 3-0;
}
else {
VX_shape1[1], 0;
VX_shape2[0], 4-2;
}
}
	4
	
	1
	3
		uimsbf
		
		uimsbf
		uimsbf

Table 2.34 — Syntax of 4k_ESC4()

Syntax	No. of bits	Mnemonic

4k_ESC4(rate)
{
if (VUV != 0) {
LSP2, 2-0;
LSP3, 3-0;
LSP5, 6-0;
Pitch, 0;
SE_shape3, 1-0;
}
else {
VX_shape2[0], 1-0;
VX_shape2[1], 4-0;
	3
	4
	7
	1
	2
	
	2
	5
		uimsbf
		uimsbf
		uimsbf
		uimsbf
		uimsbf
		
		uimsbf
		uimsbf

	VX_shape2[2], 4-0;	**5**	**uimsbf**
	if (rate >= 4000) {		
	VX_shape2[3], 4-0;	**5**	**uimsbf**
	}		
}			

2.3.4.2 Bitstream syntax for the scalable mode

Bitstream syntax of the base layer for scalable mode is the same as that of ErHVXCfixframe(2000). Bitstream syntax of enhancement layer, ErHVXCenhaFrame(), for scalable mode is shown below.

Table 2.35 — Syntax of ErHVXCenh_fixframe()

Syntax	No. of bits	Mnemonic
ErHVXCenh_fixframe()		
{		
Enh_ESC0();		
Enh_ESC1();		
Enh_ESC2();		
}		

Table 2.36 — Syntax of Enh_ESC0()

Syntax	No. of bits	Mnemonic
Enh_ESC0()		
{		
if (VUV != 0) {		
SE_shape3, 6-2;	**5**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
SE_shape4, 9;	**1**	**uimsbf**
SE_shape5, 8;	**1**	**uimsbf**
SE_shape6, 5;	**1**	**uimsbf**
SE_shape4, 8-6;	**3**	**uimsbf**
}		
else {		
LSP5, 7;	**1**	**uimsbf**
VX_gain2[0], 2-0;	**3**	**uimsbf**
VX_gain2[1], 2-0;	**3**	**uimsbf**
VX_gain2[2], 2-0;	**3**	**uimsbf**
VX_gain2[3], 2-1;	**2**	**uimsbf**
}		
}		

Table 2.37 — Syntax of Enh_ESC1()

Syntax	No. of bits	Mnemonic
Enh_ESC1()		
{		
if (VUV != 0) {		
SE_shape4, 5-0;	**6**	**uimsbf**
SE_shape5, 7-0;	**8**	**uimsbf**
SE_shape6, 4-0;	**5**	**uimsbf**
}		
else {		
VX_gain2[3], 0;	**1**	**uimsbf**

LSP5, 6-0;	**7**	**uimsbf**
VX_shape2[0], 4-0;	**5**	**uimsbf**
VX_shape2[1], 4-0;	**5**	**uimsbf**
VX_shape2[2], 4;	**1**	**uimsbf**
}		
}		

Table 2.38 — Syntax of Enh_ESC2()

Syntax	No. of bits	Mnemonic
Enh_ESC2()		
{		
if (VUV != 0) {		
LSP5, 6-0;	**7**	**uimsbf**
SE_shape3, 1-0;	**2**	**uimsbf**
}		
else {		
VX_shape2[2], 3-0;	**4**	**uimsbf**
VX_shape2[3], 4-0;	**5**	**uimsbf**
}		
}		

2.3.4.3 Bitstream syntax of variable bitrate mode

Table 2.39 — Syntax of ErHVXCvarframe()

Syntax	No. of bits	Mnemonic
ErHVXCvarframe(rate)		
{		
if (rate == 2000) {		
if (var_ScalableFlag == 1) {		
BaseVar_ESC0();		
BaseVar_ESC1();		
BaseVar_ESC2();		
BaseVar_ESC3();		
} else {		
Var2k_ESC0();		
Var2k_ESC1();		
Var2k_ESC2();		
Var2k_ESC3();		
}		
} else {		
Var4k_ESC0();		
Var4k_ESC1();		
Var4k_ESC2();		
Var4k_ESC3();		
Var4k_ESC4();		
}		
}		

Table 2.40 — Syntax of Var2k_ESC0()

Syntax	No. of bits	Mnemonic
Var2k_ESC0()		
{		

VUV, 1-0;	**2**	**uimsbf**
if (VUV == 2 || VUV == 3) {		
LSP4, 0;	**1**	**uimsbf**
SE_gain, 4-0;	**5**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
Pitch, 6-1;	**6**	**uimsbf**
LSP2, 6;	**1**	**uimsbf**
LSP3, 4;	**1**	**uimsbf**
LSP2, 5;	**1**	**uimsbf**
}		
else if (VUV == 0) {		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
LSP3, 4-3;	**2**	**uimsbf**
}		
}		

Table 2.41 — Syntax of Var2k_ESC1()

Syntax	No. of bits	Mnemonic
Var2k_ESC1()		
{		
if (VUV == 2 || VUV == 3) {		
SE_shape1, 3-0;	**4**	**uimsbf**
}		
else if (VUV == 0) {		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2;	**1**	**uimsbf**
}		
}		

Table 2.42 — Syntax of Var2k_ESC2()

Syntax	No. of bits	Mnemonic
Var2k_ESC2()		
{		
if (VUV == 2 || VUV == 3) {		
SE_shape2, 3-0;	**4**	**uimsbf**
}		
else if (VUV == 0) {		
LSP3, 1-0;	**2**	**uimsbf**
}		
}		

Table 2.43 — Syntax of Var2k_ESC3()

Syntax	No. of bits	Mnemonic
Var2k_ESC3()		
{		
if (VUV == 2 || VUV == 3) {		
LSP2, 4-0;	**5**	**uimsbf**
LSP3, 3-0;	**4**	**uimsbf**
Pitch, 0;	**1**	**uimsbf**

```
    }
}
```

Table 2.44 — Syntax of Var4k_ESC0()

Syntax	No. of bits	Mnemonic
Var4k_ESC0()		
{		
VUV,1-0;	**2**	**uimsbf**
if (VUV == 2 || VUV == 3) {		
LSP4, 0;	**1**	**uimsbf**
SE_gain,4-0;	**5**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
Pitch, 6-1;	**6**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
SE_shape3, 6-2;	**5**	**uimsbf**
LSP3, 4;	**1**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
SE_shape4, 9;	**1**	**uimsbf**
SE_shape5, 8;	**1**	**uimsbf**
SE_shape6, 5;	**1**	**uimsbf**
}		
else if (VUV == 0) {		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
LSP3, 4-3;	**2**	**uimsbf**
}		
else {		
UpdateFlag, 0;	**1**	**uimsbf**
if (UpdateFlag == 1) {		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
LSP3, 4-3;	**2**	**uimsbf**
}		
}		
}		

Table 2.45 — Syntax of Var4k_ESC1()

Syntax	No. of bits	Mnemonic
Var4k_ESC1()		
{		
if (VUV == 2 || VUV == 3) {		
SE_shape4, 8-0;	**9**	**uimsbf**
SE_shape5, 7-0;	**8**	**uimsbf**
SE_shape6, 4-0;	**5**	**uimsbf**
}		
else if (VUV == 0) {		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2-0;	**3**	**uimsbf**
VX_shape1[0], 5-0;	**6**	**uimsbf**
VX_shape1[1], 5-0;	**6**	**uimsbf**

}
else {
if (UpdateFlag == 1) {
LSP2, 2-0;
LSP3, 2-0;
}
}
}
	3
	3
		uimsbf
		uimsbf

Table 2.46 — Syntax of Var4k_ESC2()

Syntax	No. of bits	Mnemonic

Var4k_ESC2()
{
if (VUV == 2 || VUV == 3) {
SE_shape1, 3-0;
}
}
	4	**uimsbf**

Table 2.47 — Syntax of Var4k_ESC3()

Syntax	No. of bits	Mnemonic

Var4k_ESC3()
{
if (VUV == 2 || VUV == 3) {
SE_shape2, 3-0;
}
}
	4	**uimsbf**

Table 2.48 — Syntax of Var4k_ESC4()

Syntax	No. of bits	Mnemonic

Var4k_ESC4()
{
if (VUV == 2 || VUV == 3) {
LSP2, 2-0;
LSP3, 3-0;
LSP5, 6-0;
Pitch, 0;
SE_shape3, 1-0;
}
}
	3
	4
	7
	1
	2
		uimsbf
		uimsbf
		uimsbf
		uimsbf
		uimsbf

Table 2.49 — Syntax of BaseVar_ESC0()

Syntax	No. of bits	Mnemonic

BaseVar_ESC0()
{
VUV, 1-0;
if (VUV == 2 || VUV == 3) {
LSP4, 0;
SE_gain, 4-0;
LSP1, 4-0;
Pitch, 6-1;
LSP2, 6;
	2
	1
	5
	5
	6
	1
		uimsbf
		uimsbf
		uimsbf
		uimsbf
		uimsbf
		uimsbf

LSP3, 4;	**1**	**uimsbf**
LSP2, 5;	**1**	**uimsbf**
}		
else if (VUV == 0) {		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
LSP3, 4-3;	**2**	**uimsbf**
}		
else {		
UpdateFlag, 0;	**1**	**uimsbf**
if (UpdateFlag == 1) {		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
LSP3, 4-3;	**2**	**uimsbf**
}		
}		
}		

Table 2.50 — Syntax of BaseVar_ESC1()

Syntax	No. of bits	Mnemonic
BaseVar_ESC1()		
{		
if (VUV == 2 || VUV == 3) {		
SE_shape1, 3-0;	**4**	**uimsbf**
}		
else if (VUV == 0) {		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2;	**1**	**uimsbf**
}		
else {		
if (UpdateFlag == 1) {		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2-0;	**3**	**uimsbf**
}		
}		
}		

Table 2.51 — Syntax of BaseVar_ESC2()

Syntax	No. of bits	Mnemonic
BaseVar_ESC2()		
{		
if (VUV == 2 || VUV == 3) {		
SE_shape2, 3-0;	**4**	**uimsbf**
}		
else if (VUV == 0) {		
LSP3, 1-0;	**2**	**uimsbf**
VX_shape1[0], 5-4;	**2**	**uimsbf**
}		
}		

Table 2.52 — Syntax of BaseVar_ESC3()

Syntax	No. of bits	Mnemonic
BaseVar_ESC3()		
{		
if (VUV == 2 || VUV == 3) {		
LSP2, 4-0;	**5**	**uimsbf**
LSP3, 3-0;	**4**	**uimsbf**
Pitch, 0;	**1**	**uimsbf**
}		
else if (VUV == 0) {		
VX_shape1[0], 3-0;	**4**	**uimsbf**
VX_shape1[1], 5-0;	**6**	**uimsbf**
}		
}		

2.3.4.4 Enhancement Layer of the scalable mode of the variable bitrate mode

Table 2.53 — Syntax of ErHVXCenh_varframe()

Syntax	No. of bits	Mnemonic
ErHVXCenh_varframe()		
{		
EnhVar_ESC0();		
EnhVar_ESC1();		
EnhVar_ESC2();		
}		

Table 2.54 — Syntax of EnhVar_ESC0()

Syntax	No. of bits	Mnemonic
EnhVar_ESC0()		
{		
if (VUV == 2 || VUV == 3) {		
SE_shape3, 6-2;	**5**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
SE_shape4, 9;	**1**	**uimsbf**
SE_shape5, 8;	**1**	**uimsbf**
SE_shape6, 5;	**1**	**uimsbf**
SE_shape4, 8-6;	**3**	**uimsbf**
}		
}		

Table 2.55 — Synatx of EnhVar_ESC1()

Syntax	No. of bits	Mnemonic
EnhVar_ESC1()		
{		
if (VUV == 2 || VUV == 3) {		
SE_shape4, 5-0;	**6**	**uimsbf**
SE_shape5, 7-0;	**8**	**uimsbf**
SE_shape6, 4-0;	**5**	**uimsbf**
}		
}		

Table 2.56 — Syntax of EnhVar_ESC2()

Syntax	No. of bits	Mnemonic
EnhVar_ESC2()		
{		
if (VUV == 2 || VUV == 3) {		
LSP5, 6-0;	**7**	**uimsbf**
SE_shape3, 1-0;	**2**	**uimsbf**
}		
}		

2.4Bitstream semantics

2.4.1 Decoder configuration (HvxcSpecificConfig,ErrorResilientHvxcSpecificConfig)

HVXCvarMode: A flag indicating HVXC variable rate mode(Table 2.1).

HVXCrateMode: A 2 bit field indicating HVXC bit rate mode(Table 2.1).

extensionFlag: A flag indicating the presence of MPEG-4 Version 2 data (Table 2.1).

var_ScalableFlag: A flag indicating ER_HVXC variable scalable mode(Table 2.22).

isBaseLayer: A one-bit identifier representing whether the corresponding layer is the base layer (1) or the enhancement layer (0).

2.4.2 Bitstream frame (alPduPayload)

LSP1: This 5 bits field represents the index of the first stage LSP quantization (base layer, Table 2.8 and Table 2.17).

LSP2: This 7 bits field represents the index of the second stage LSP quantization (base layer, Table 2.8 and Table 2.17).

LSP3: This 5 bits field represents the index of the second stage LSP quantization (base layer, Table 2.8 and Table 2.17).

LSP4: This 1 bit field represents the flag whether a interframe prediction is used or not in the second stage LSP quantization (base layer, Table 2.8 and Table 2.17).

LSP5: This 8 bits field represents the index of the third stage LSP quantization (enhancement layer, Table 2.9).

VUV: This 2 bits field represents V/UV decision mode. It should be noted that this field has a different meaning according to HVXC variable rate mode(Table 2.10 and Table 2.15)

Pitch: This 7 bits field represents the index of the linearly quantized pitch lag ranging from 20 to 147 samples(Table 2.12 and Table 2.18).

SE_shape1: This 4 bits field represents the index of the spectral envelope shape (base layer, Table 2.12 and Table 2.18).

SE_shape2: This 4 bits field represents the index of the spectral envelope shape (base layer, Table 2.12 and Table 2.18).

SE_gain: This 5 bits field represents the index of the spectral envelope gain (base layer, Table 2.12 and Table 2.18).

VX_shape1[sf_num]: This 6 bits field represents the index of the sf_num-th subframe's VXC shape (base layer, Table 2.12 and Table 2.18).

VX_gain1[sf_num]: This 4 bits field represents the index of the sf_num-th subframe's VXC gain (base layer, Table 2.12 and Table 2.18).

SE_shape3: This 7 bits field represents the index of the spectral envelope shape (enhancement layer, Table 2.13).

SE_shape4: This 10 bits field represents the index of the spectral envelope shape (enhancement layer, Table 2.13).

SE_shape5: This 9 bits field represents the index of the spectral envelope shape (enhancement layer, Table 2.13).

SE_shape6: This 6 bits field represents the index of the spectral envelope shape (enhancement layer, Table 2.13).

VX_shape2[sf_num]: This 5 bits field represents the index of the sf_num-th subframe's VXC shape (enhancement layer, Table 2.13).

VX_gain2[sf_num]: This 3 bits field represents the index of the sf_num-th subframe's VXC gain (enhancement layer, Table 2.13).

UpdateFlag: This 1 bit field represents a flag to indicate update noise frame(only for ER_HVXC 4 kbit/s variable rate mode).

2.5HVXC decoder tools

2.5.1 Overview

HVXC provides an efficient coding scheme for Linear Predictive Coding (LPC) residuals based on harmonic and stochastic vector representation. Vector quantization (VQ) of the spectral envelope of LPC residuals with a weighted distortion measure is used when the signal is voiced. Vector excitation coding (VXC) is used when the signal is unvoiced. The major algorithmic features are:

- Weighted VQ of variable dimension spectral vector.
- A fast harmonic synthesis algorithm by IFFT.
- Interpolative coder parameters for speed/pitch control

Also, functional features include:

- As low as 33.5 ms of total algorithmic delay
- 2.0-4.0 kbps scalable mode
- Variable bit rate coding for rates less than 2.0 kbps

2.5.1.1 Framing structure and block diagram of the decoder

Figure 2.2 shows the overall structure of the HVXC decoder. The HVXC decoder tools allow decoding of speech signals at 2 kbit/s and higher, up to 4 kbit/s. HVXC decoder tools also allow decoding with variable bit rate mode at an average bit rate of around 1.2-1.7 kbps. The basic decoding process is composed of four steps; de-quantization of parameters, generation of excitation signals for voiced frames by sinusoidal synthesis (harmonic synthesis) and noise component addition, generation of excitation signals for unvoiced frames by codebook look-up, and LPC synthesis. To enhance the synthesized speech quality spectral post-filtering is used.

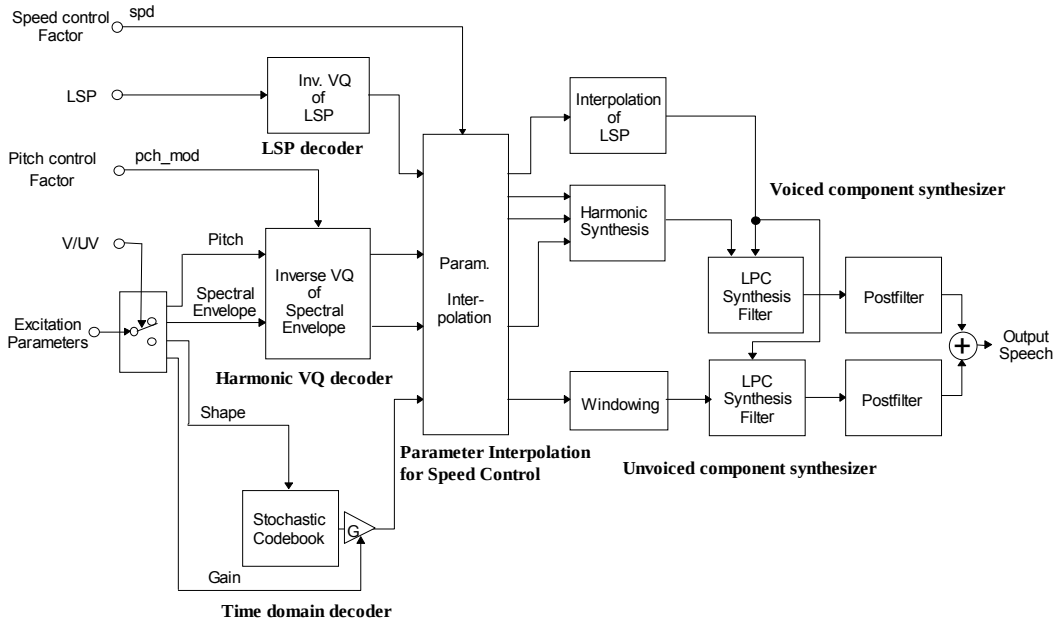


Figure 2.2 — Block diagram of the HVXC decoder

2.5.1.2 Delay mode

HVXC coder/decoder supports low/normal delay encode/decode mode, allowing any combinations of delay mode at 2.0-4.0 kbps with a scalable scheme. The figure below shows the framing structure of each delay mode. The frame length is 20 ms for all the delay modes. For example, use of low delay encode and low delay decode mode results in a total coder/decoder delay of 33.5 ms.

In the encoder, the algorithmic delay could be selected to be either 26 ms or 46 ms. When 46 ms delay is selected, one frame look ahead is used for pitch detection. When 26 ms delay is selected, only the current frame is used for pitch detection. For both cases, syntax is common, all the quantizers are common, and bitstreams are compatible. In the decoder the algorithmic delay can be selected to be either 10 ms (normal delay mode) or 7.5 ms (low delay mode). When 7.5 ms delay is selected, the decoder frame interval is shifted by 2.5 ms (20 samples) compared to the 10 ms delay mode. In this case, the excitation generation and LPC synthesis phase is shifted by 2.5 ms. Again, for both cases, syntax is common, all the quantizers are common, and bitstreams are compatible.

In summary, any independent choice of encoder/decoder delay from the following combination is possible:

Encoder delay: 26 ms or 46 ms

Decoder delay: 10 ms or 7.5 ms

One or some combinations of the delay mode shall be supported depending on the application.

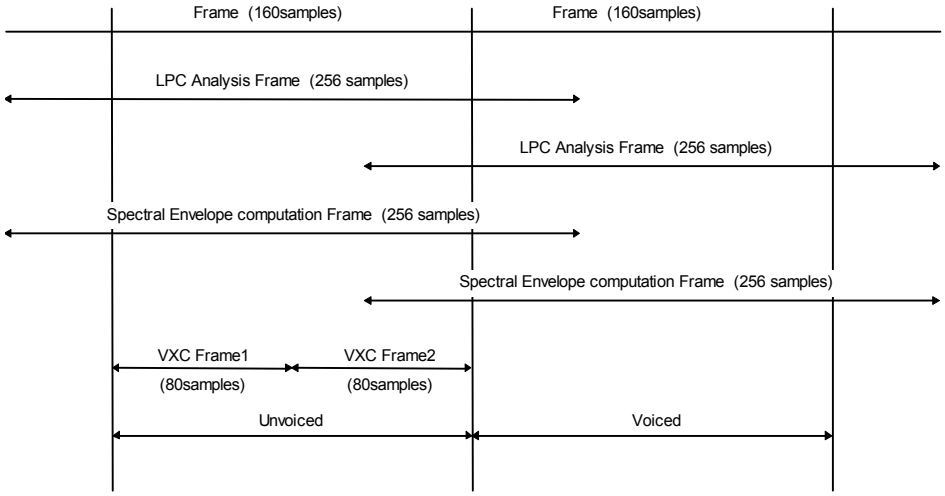


Figure 2.3 — Framing structure of HVXC

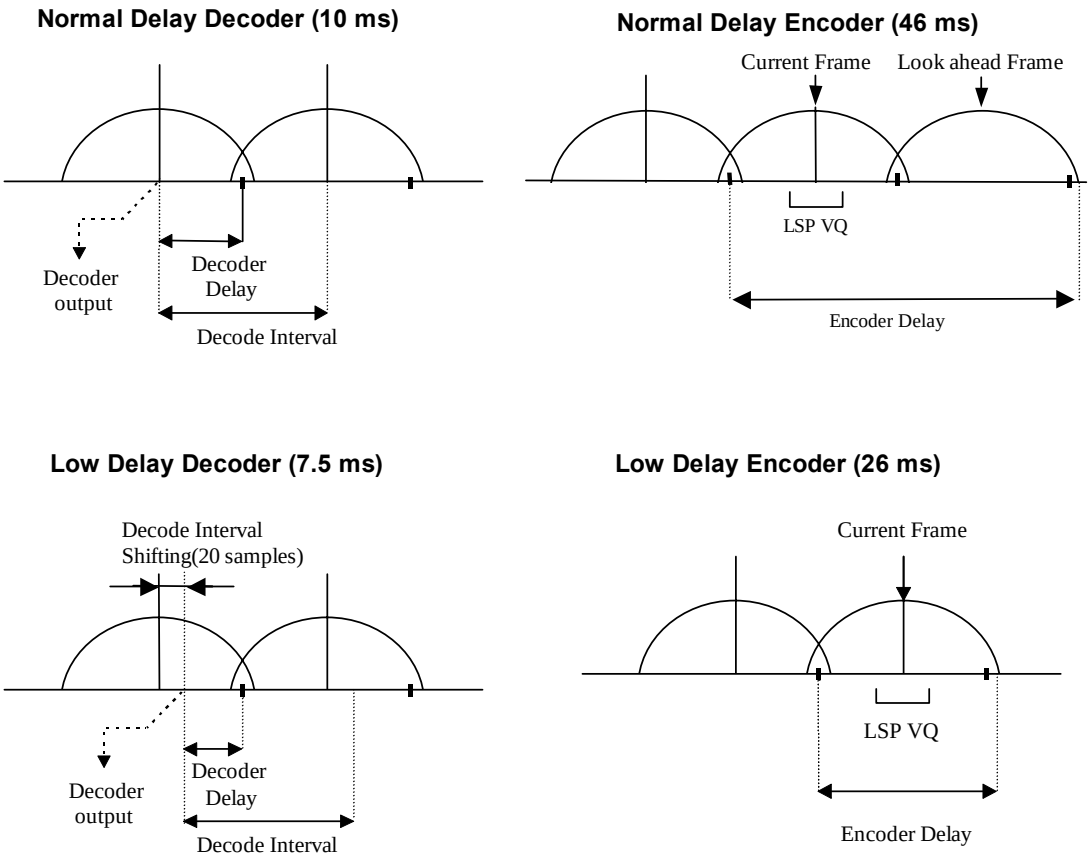


Figure 2.4 — Delay mode of the HVXC encoder and decoder

2.5.2 LSP decoder

2.5.2.1 Tool description

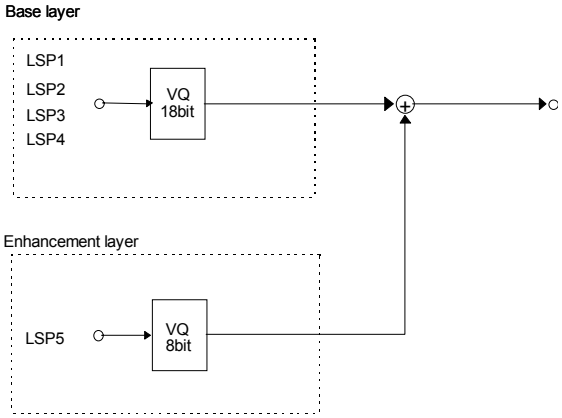


Figure 2.5 — LSP decoder

For the quantization of the LSP parameters, a multistage quantizer structure is used and the output vectors from each stage have to be summed up to obtain the LSP parameters.

When the bitrate is 2 kbps, the LSPs of the current frame, which are coded by split and two-stage vector quantization, are decoded using a two-stage decoding process. At 4 kbps, a 10-dimensional vector quantizer, which has an 8 bit codebook, is added to the bottom of the LSP quantizer scheme of 2.0 kbps coder. The bits needed for the LSPs are increased from 18bits/20msec to 26bits/20msec.

Table 2.57 — Configuration of the multistage LSP VQ

1st stage	10 LSP VQ	5 bits
2nd stage	(5+5) LSP VQ	(7+5+1) bits
3rd stage	10 LSP VQ	8 bits

2.5.2.2 Definitions

Definitions of constants

$LPCORDER$: LPC analysis order (=10)

$dim[][]$: dimensions for the split vector quantization

min_gap : minimum distance between adjacent LSP coefficients (base layer, =4.0/256.0)

$ratio\_predict$: LSP interframe prediction ratio (=0.7)

$THRSLD\_L$: minimum distance between adjacent LSP coefficients

(low frequency part of enhancement layer, =0.020)

$THRSLD\_M$: minimum distance between adjacent LSP coefficients

(middle frequency part of enhancement layer, =0.020)

$THRSLD\_H$: minimum distance between adjacent LSP coefficients

(high frequency part of enhancement layer, = 0.020)

Definitions of variables

$qLsp[]$: quantized LSP parameters

$LSP1$: the index of the first stage LSP quantization (base layer)

$LSP2$, $LSP3$: the indices of the second LSP quantization (base layer)

$LSP4$: the flag showing whether a interframe prediction is used or not (base layer)

$LSP5$: the index of the third LSP quantization (enhancement layer)

$lsp\_tbl[][][]$: look-up tables for the first stage decoding process
$d\_tbl[][][]$: look-up tables for the second stage decoding process of the VQ without interframe prediction

$pd\_tbl[][][]$: look-up tables for the second stage decoding process of the VQ with interframe prediction.

$vqLsp[][]$: look-up table for the enhancement layer
$sign$: sign of code vector for the second stage decoding process
idx : unpacked index for the second stage decoding process
$lsp\_predict[]$: the LSPs predicted from $lsp\_previous[]$ and $lsp\_first[]$
$lsp\_previous[]$: the LSPs decoded at the previous frame
$lsp\_current[]$: the LSPs decoded at the current frame
$lsp\_first[]$: the LSPs decoded at the first stage decoding process

2.5.2.3 Decoding process

The decoding process of the LSP parameters for the base layer (2.0 kbps) is the same as that of the narrowband CELP. The decoding process is as described below.

Converting indices to LSPs

The LSPs of the current frame ($lsp\_current[]$), which are coded by split and two-stage vector quantization, are decoded with a two-stage decoding process. The dimension of each vector is described in the tables below. The **LSP1** and **LSP2, LSP3** represent indices for the first and the second stage respectively.

Table 2.58 — Dimension of the first stage LSP vector

Split Vector Index: i	Vector Dimension: $dim[0][i]$
0	10

Table 2.59 — Dimension of the second stage LSP vector

Split Vector Index: i	Vector Dimension: $dim[1][i]$
0	5
1	5

In the first stage, the LSP vector of the first stage $lsp\_first[]$ is decoded simply by looking up the table $lsp\_tbl[][][]$. (The table $lsp\_tbl[][][]$ is shown in Annex 2.E)

```
for (i = 0; i < dim[0][0]; i++) {
   lsp_first[i] = lsp_tbl[0][LSP1][i];
}
```

In the second stage, there are two types of decoding processes, namely, decoding process of VQ without interframe prediction and VQ with interframe prediction. The flag **LSP4** indicates which process should be selected.

Table 2.60 — Decoding process for the second stage

LSP Index: **LSP4**	Decoding process
0	VQ without interframe prediction
1	VQ with interframe prediction

Decoding process of VQ without interframe prediction

In order to obtain LSPs of the current frame $lsp\_current[]$, the decoded vectors in the second stage are added to the decoded first stage LSP vector $lsp\_first[]$. The MSB of the **LSP2** and **LSP3** represents the sign of the decoded vector, and the remaining bits represent the index for the table $d\_tbl[][][]$. (The table $d\_tbl[][][]$ is shown in Annex 2.E)

```
sign = LSP2>>6;
idx = LSP2&0x3f;
if (sign == 0) {
   for (i = 0; i < dim[1][0]; i++) {
      lsp_current[i] = lsp_first[i] + d_tbl[0][idx][i];
   }
}
else {
   for (i = 0; i < dim[1][0]; i++) {
      lsp_current[i] = lsp_first[i] - d_tbl[0][idx][i];
   }
}
sign = LSP3>>4;
idx = LSP3&0x0f;
if (sign == 0) {
   for (i = 0; i < dim[1][1]; i++) {
      lsp_current[dim[1][0]+i] = lsp_first[dim[1][0]+i] + d_tbl[1][idx][i];
   }
}
else {
   for (i = 0; i < dim[1][1]; i++) {
      lsp_current[dim[1][0]+i] = lsp_first[dim[1][0]+i] - d_tbl[1][idx][i];
   }
}
```

Decoding process of VQ with interframe prediction

In order to obtain LSPs of the current frame $lsp\_current[]$, the decoded vectors of the second stage are added to the LSP vector $lsp\_predict[]$ which are predicted from the decoded LSPs of the previous frame $lsp\_previous[]$ and the decoded first stage LSP vector $lsp\_first[]$. As with the decoding process of VQ without interframe prediction, the MSB of the **LSP2** and **LSP3** represents the sign of the decoded vector, and the remaining bits represent the index for the table $pd\_tbl[][][]$. (The table $pd\_tbl[][][]$ is shown in Annex 2.E)

```
for (i = 0; i < LPCORDER; i++) {
   lsp_predict[i] = (1-ratio_predict)*lsp_first[i]
                      + ratio_predict*lsp_previous[i];
}


sign = LSP2>>6;
idx = LSP2&0x3f;
if (sign == 0) {
   for (i = 0; i < dim[1][0]; i++) {
      lsp_current[i] = lsp_predict[i] + pd_tbl[0][idx][i];
   }
}
else {
   for (i = 0; i < dim[1][0]; i++) {
      lsp_current[i] = lsp_predict[i] - pd_tbl[0][idx][i];
   }
}
sign = LSP3>>4;
idx = LSP3&0x0f;
if (sign == 0) {
   for (i = 0; i < dim[1][1]; i++) {
      lsp_current[dim[1][0]+i] = lsp_predict[dim[1][0]+i] + pd_tbl[1][idx][i];
   }
}
else {
   for (i = 0; i < dim[1][1]; i++) {
      lsp_current[dim[1][0]+i] = lsp_predict[dim[1][0]+i] - pd_tbl[1][idx][i];
   }
}
```

Stabilization of LSPs

The decoded LSPs *lsp_current*[] are stabilized in order to ensure stability of the LPC synthesis filter which is derived from the decoded LSPs. The decoded LSPs are arranged in ascending order, having a minimum distance between adjacent coefficients.

```
for (i = 0; i < LPCORDER; i++) {
   if (lsp_current[i] < min_gap) lsp_current[i] = min_gap;
}
for (i = 0; i < LPCORDER-1; i++) {
   if (lsp_current[i+1]-lsp_current[i] < min_gap) {
      lsp_current[i+1] = lsp_current[i]+min_gap;
   }
}
for (i = 0; i < LPCORDER; i++) {
   if (lsp_current[i] > 1-min_gap) lsp_current[i] = 1-min_gap;
}
for (i = LPCORDER-1; i > 0; i--) {
   if (lsp_current[i]-lsp_current[i-1] < min_gap) {
      lsp_current[i-1] = lsp_current[i]-min_gap;
   }
}


for (i = 0; i < LPCORDER; i++) {
   qLsp[i] = lsp_current[i];
}
```

Storing the coefficients

After the LSP decoding process, the decoded LSPs have to be stored in memory, since they are used for prediction at the next frame.

```
for (i = 0; i < LPCORDER; i++) {
   lsp_previous[i] = lsp_current[i];
}
```

It must be noted that the stored LSPs *lsp_ previous*[] must be initialized as described below when the whole decoder is initialized.

```
for (i = 0; i < LPCORDER; i++) {
   lsp_previous[i] = (i+1)/(LPCORDER+1);
}
```

Decoding process for the enhancement layer

For the enhancement layer (4.0 and 3.7 kbps), additional code vectors and the base layer's LSPs are summed up as follows.

```
for (i = 0; i < LPCORDER; i++) {
  qLsp[i] += vqLsp[LSP5][i];
}
```

After the calculation, the LSPs are stabilized again.

```
for (i = 0; i < 2; i++) {
   if (qLsp[i+1] - qLsp[i] < 0) {
      tmp = qLsp[i+1];
      qLsp[i+1] = qLsp[i];
      qLsp[i] = tmp;
   }
   if (qLsp[i+1] - qLsp[i] < THRSLD_L) {
      qLsp[i + 1] = qLsp[i] + THRSLD_L;
   }
}

for (i = 2; i < 6; i++) {
   if (qLsp[i+1] - qLsp[i] < THRSLD_M) {
      tmp = (qLsp[i+1] + qLsp[i])/2.0;
      qLsp[i+1] = tmp + THRSLD_M/2.0;
      qLsp[i] = tmp - THRSLD_M/2.0;
   }
}

for (i = 6; i < LPCORDER-1; i++) {
   if (qLsp[i+1] - qLsp[i] < 0) {
      tmp = qLsp[i+1];
      qLsp[i+1] = qLsp[i];
      qLsp[i] = tmp;
   }
   if (qLsp[i+1] - qLsp[i] < THRSLD_H) {
      qLsp[i] = qLsp[i+1] - THRSLD_H;
   }
}
```

2.5.2.4 Tables

See LSP quantizer table of Annex 2.E.

2.5.3 Harmonic VQ decoder

2.5.3.1 Tool description

The decoding process consists of two major steps for the base layer, that is, inverse vector quantization of spectral envelope vectors and dimension conversion. For the enhancement layer, additional inverse-quantizers are used . The operations of each step are given below.

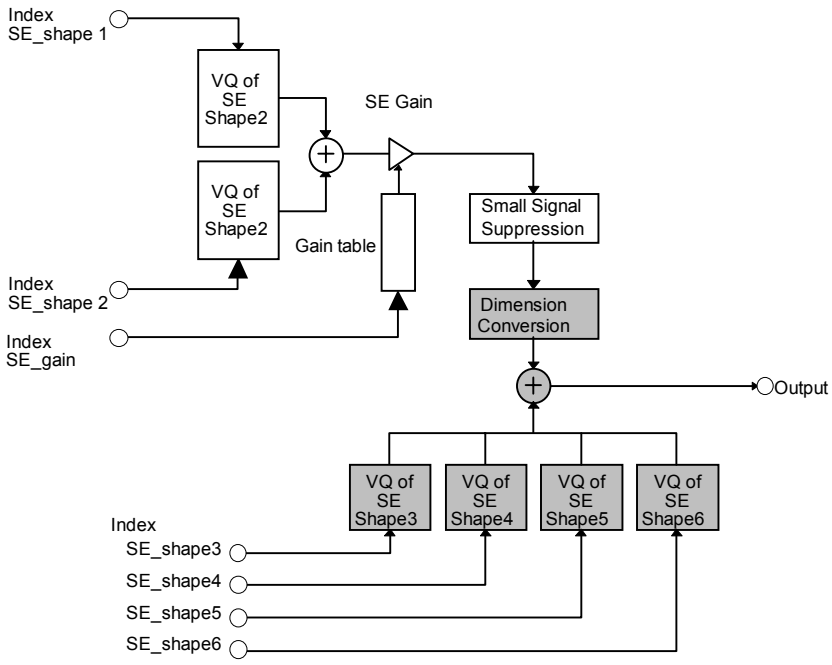


Figure 2.6 — Structure of Harmonic VQ decoder

2.5.3.2 Definitions

Definitions of constants

$SAMPLE$: the number of samples in frequency spectrum ranging from 0 to 2π (=256)

R : over sampling rate in the dimension conversion (=8)

$vqdim0$: vector dimension of spectral envelope quantization (=44)

$JISU$: order of the over sampling filter in up sampled domain (=9)

$f\_coef[]$: over sampling filter coefficients

Definition of variables

$Pitch$: the index of linearly quantized pitch lag value

pch : pitch lag value in the current frame

pch_mod : pitch modification factor

$w0f$: the original fundamental frequency where $SAMPLE$ represents 2π

$send$: the number of harmonics in the current frame (between 0 and 3800Hz)

$w0$: target fundamental frequency after the dimension conversion where $SAMPLE \times R$ represents 2π

HVXCrate : operating bit rate of the decoder

$qedvec[]$: quantized spectral envelope vector in the fixed dimension

SE_gain : the index of spectral envelope gain (base layer)

SE_shape1, SE_shape2 : the indices of spectral envelope shape (base layer)

$SE\_shape3, SE\_shape4, SE\_shape5, SE\_shape6$: the indices of spectral envelope shape (enhancement layer)

$g0[]$: SE_gain codeword

$cb0[][]$: SE_shape1 codeword

$cb1[][]$: SE_shape2 codeword

$cb4k[0][][]$: SE_shape3 codeword

$cb4k[1][][]$: SE_shape4 codeword

$cb4k[2][][]$: SE_shape5 codeword

$cb4k[3][][]$: SE_shape6 codeword

$re[]$: input of the dimension conversion

$rel0, rel1$: 8 times over sampled values in the dimension conversion

ip_ratio : linear interpolation ratio of the dimension conversion

$target[]$: reconstructed vector due to the vector quantizers for the enhancement layer

$am[]$: reconstructed harmonic magnitudes vector

$feneq$: rms of quantized spectral envelope vector in the current frame

$feneqold$: rms of quantized spectral envelope vector in the previous frame

2.5.3.3 Decoding process

Pitch index decoding

The pitch lag value in the current frame, pch , is decoded from the pitch index **Pitch** as follows:

$$pch = Pitch + 20.0$$

Pitch modification can be done by dividing pch by pitch modification factor pch_mod :

$$pch = pch / pch\_mod$$

If the pitch modification is controlled by the **pitch** field in the AudioSource BIFS node, the modification factor is:

$$pch\_mod = pitch$$

It should be noted that the modulated pitch lag value must be within a range from 8.0 to 147.0.

Then the number of harmonics in a frequency range between 0 and 3800Hz, $send$,and the fundamental frequency, $w0$ (where $SAMPLE \times R$ represents 2π), are calculated as follows:

$$send = (\text{int})(0.95 \times pch \times 0.5)$$

$$w0 = \frac{SAMPLE \times R}{pch}$$

Harmonic magnitudes decoding

Decoding of harmonic magnitudes consists of the following steps;

(S1) Inverse vector quantization of the base layer

(S2) Suppression of small signals

(S3) Dimension conversion of base layer output

(S4) Inverse vector quantization of the enhancement layer

For the 2.0kbps mode, **S1**, **S2** and **S3** above are executed to obtain the harmonic magnitudes. For the 4.0 and 3.7kbps mode, **S4** is executed in addition to **S1**, **S2** and **S3**.

In the 2.0 kbps mode, a combination of two-stage shape vector quantization and a scalar gain quantization is used, whose indices are **SE_shape1**, **SE_shape2** and **SE_gain** respectively. The dimension of the two shape codebooks is fixed (=44). In **S1**, two shape vectors represented by **SE_shape1** and **SE_shape2** are added and then multiplied by the gain represented by **SE_gain**. The spectral envelope vector obtained in **S1** covers frequency range from 0 to 3800 Hz. The spectral envelope vector of very small energy is then suppressed in **S2**. In order to obtain the harmonic magnitudes vector of the original dimension, $send$, dimension conversion is then applied to the spectral envelope vector in **S3**. In the 4.0 kbps mode, additional stage with a split VQ scheme composed of four vector quantizers is used for the enhancement layer. **SE_shape3**, **SE_shape4**, **SE_shape5** and **SE_shape6** represents the indices of the quantizers for the enhancement layer. In **S4**, the output of these quantizers is added to the output of the **S3** for harmonic magnitudes at the lowest 14 harmonic frequencies. When 3.7kbps mode is selected, **SE_shape6** is not available and **S4** is carried out for the harmonic magnitudes of only the lowest 10 harmonic frequencies.

Table 2.61 — Configuration of the multistage harmonic VQ

2.0 kbps two-stage VQ	4bits shape + 4bits shape + 5bits gain

dimension	44			
4.0 kbps	7bits	10bits	9bits	6bits
split VQ dimension	2	4	4	4

Dimension conversion algorithm

The theoretical background of the dimension conversion algorithm used in this tool is explained below.

The number of points which composes the spectral envelope varies depending on the pitch value, since the spectral envelope is a set of the estimates of the magnitudes at each harmonic. The number of harmonics ranges from 9 to 70. In order to obtain the variable number of harmonic magnitudes, the decoder has to convert a fixed-dimension (=44) codevector to a variable dimension vector. The number of points, which represent the shape of the spectral envelope, should be modified without changing the shape. For this purpose, a dimension converter consisting of a combination of a low pass filter and a 1st order linear interpolator is used. An FIR low pass filter with 7 sets of coefficients, each set consisting of 8 coefficients, is used for the first stage 8-times over-sampling. The 7 sets of the filter coefficients are obtained by grouping 8 every coefficients from a windowed sinc, $f\_coef[i]$, with the offsets of 1 through 7, where

$$f\_coef[i] = \frac{\sin \pi(i-32)/8}{\pi(i-32)/8}\left(0.5 - 0.5\cos 2\pi i/64\right) \qquad 0 \le i \le 64$$

This FIR filtering allows decimated computation, in which only the points used at the next stage are computed. They are the left and right adjacent points of the final output of the dimension converter.

At the second over-sampling stage, 1st order linear interpolation is applied to obtain the necessary output points. In this way, harmonic magnitudes vectors of variable-dimension are obtained from spectral envelope vectors of fixed dimension (=44).

(S1) Inverse vector quantization of the base layer:

```
qedvec[0] = 0.0f;
for (i = 0; i < vqdim0; i++)
   qedvec[i+1] = g0[SE_gain]*(cb0[SE_shape1][i]+cb1[SE_shape2][i]);
```

(S2) Small signal suppression:

```
feneq = 0.0f;
for (i = 0; i < vqdim0; i++)
   feneq += qedvec[i+1]*qedvec[i+1];
feneq = sqrt(feneq/(float)vqdim0);
if (feneq < 1.0f || 0.5f*(feneqold+feneq) < 1.4f) {
   for (i = 0; i < vqdim0; i++) qedvec[i+1] = 0.0f;
}
feneqold = feneq;
```

(S3) Dimension conversion of the base layer output:

```
for (i = 0; i < (JISU-1)/2; i++)
     re[i] = 0.0f;
  for (i = 0; i <= vqdim0; i++)
     re[i+(JISU-1)/2] = qedvec[i];
  for (i = 0; i < (JISU-1)/2; i++)
     re[i+vqdim0+1+(JISU-1)/2] = qedvec[vqdim0];

  w0f = (float)(SAMPLE*0.5*0.95)/(float)vqdim0;
```

```
ii = 0;
for (i = 0; i <= vqdim0 && ii <= send; i++) {
   for (p = 0; p < R && ii <= send; p++) {
      ip_ratio = (i*R+p+1)*w0f-w0*ii;
      if (ip_ratio > 0) {
      ip_ratio /= w0f;
      rel0 = rel1 = 0.0f;
      for (j = 1; j < JISU; j++) {
         rel0 += f_coef[j*R-p]*re[i+j];
         rel1 += f_coef[j*R-(p+1)]*re[i+j];
      }
      am[ii] = rel0*ip_ratio + rel1*(1.0f-ip_ratio);
      ii++;
   }
}
```

(S4) Inverse vector quantization of the enhancement layer:

```
target[0]=0.;
k = 1;
for (i = 0; i < 2; i++, k++)
   target[k] = cb4k[0][SE_shape3][i];
for (i = 0; i < 4; i++, k++)
   target[k] = cb4k[1][SE_shape4][i];
for (i = 0; i < 4; i++, k++)
   target[k] = cb4k[2][SE_shape5][i];
if(HVXCrate >= 4000){
   for (i = 0; i < 4; i++, k++)
      target[k] = cb4k[3][SE_shape6][i];
}
else{
   for (i = 0; i < 4; i++, k++)
      target[k] = 0.0f;
}
if (send > 14) {
   for (i = 15; i <= send; i++)
      target[i] = 0.0f;
}

for (i = 0; i <= send; i++)
   am[i] += target[i];
```

2.5.3.4 Tables

See harmonic VQ table of Annex 2.E

2.5.4 Time domain decoder

2.5.4.1 Tool description

For unvoiced segments of speech, a scheme that is similar to VXC (Vector Excitation Coding) is used. The time domain decoder generates an excitation waveform for the unvoiced portion by table look up using the transmitted indices. The shape vector and gain of the base layer are updated every 10 ms. The shape is scaled by multiplying each sample with the gain value. In the 2.0 kbps mode, only the output of the first stage (base layer) is used. In the 4.0 kbps and 3.7 kbps modes, the shape vector and the gain of the second stage (enhancement layer) are multiplied and added to the output of the first stage. The shape and gain of the enhancement layer are updated every 5 ms.

Table 2.62 — Configuration of the time domain VQ

1st stage	(80dimension 6bits shape + 4bits gain) x 2
2nd stage	(40dimension 5bits shape + 3bits gain) x 4

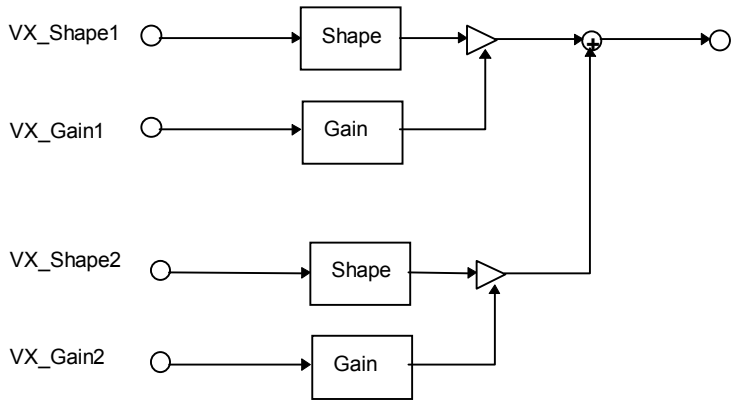


Figure 2.7 — Time domain VQ decoder

2.5.4.2 Definitions

Definitions of constants

$DimShape$: the first stage VXC frame length (=80)

$DimShape2$: the second stage VXC frame length (=40)

Definition of variables

HVXCrate : operating bit rate of the decoder

$res[i]$: VXC decoder output ($0 \le i < FRM$)

$cbL0\_g[i]$: i -th entry of the first stage VXC gain codebook

$cbL0\_s[i][j]$: j -th component of i -th entry of the first stage VXC shape codebook

$cbL1\_g[i]$: i -th entry of the second stage VXC gain codebook

$cbL1\_s[i][j]$: j -th component of i -th entry of the second stage VXC shape codebook

$VX\_gain1[i]$: the index of the i -th subframe's VXC gain (base layer, $i = 0,1$)

$VX\_shape1[i]$: the index of the i -th subframe's VXC shape (base layer, $i = 0,1$)

$VX\_gain2[i]$: the index of the i -th subframe's VXC gain (enhancement layer, $i = 0,1,2,3$)

$VX\_shape2[i]$: the index of the i -th subframe's VXC shape (enhancement layer, $i = 0,1,2,3$)

2.5.4.3 Decoding process

For the base layer:

```
for (i = 0; i < DimShape; i++)
   res[i] = cbL0_g[VX_gain[0]] * cbL0_s[VX_shape[0]][i];
for (i = 0; i < DimShape; i++)
   res[i + DimShape] = cbL0_g[VX_gain[1]] *cbL0_s[VX_shape[1]][i];
```

To add the enhancement layer:

```
if(HVXCrate >= 4000){   /* 4.0 kbps mode */
   for (i = 0; i < 4; i++){
      for (j = 0; j < DimShape2; j++){
         res[j+DimShape2*i] += cbL1_g[VX_gain2[i]]*cbL1_s[VX_shape2[i]][j];
      }
   }
}
else{                   /* 3.7kbps mode */
   for (i = 0; i < 3; i++){
      for (j = 0; j < DimShape2; j++){
         res[j+DimShape2*i] += cbL1_g[VX_gain2[i]]*cbL1_s[VX_shape2[i]][j];
      }
   }
}
```

2.5.4.4 Tables

See stochastic codebook table of Annex 2.E.

2.5.5 Parameter interpolation for speed control

2.5.5.1 Tool description

The scheme for Speed Control is explained in this subclause. The decoder has a scheme for parameter interpolation to generate the input for the “Voiced Component Synthesizer” and “Unvoiced Component Synthesiszer” at any arbitrary time instant. By this scheme, a sequence of parameters in modified intervals are computed and applied to the both of the synthesizers. In this way, decoder output in a modified time scale is obtained.

2.5.5.2 Definitions

Definitions of constants

FRM : frame interval (=160)

P : LPC order (=10)

Definitions of variables

"Parameter Interpolation" block computes the parameters in the modified time scale by interpolating the received parameters. The operation of this block is described below.

The operation of this block is basically linear interpolation and replacement of parameters.

Let us denote the arrays of original parameters as:

$pch[n]$: pitch lag value at time index n

$vuv[n]$: V/UV index at time index n

$lsp[n][i]$: decoded LSPs at time index n ($0 \le i < P$)

$send[n]$: number of harmonics at time index n

$am[n][i]$: harmonic magnitudes at time index n ($0 \le i < send[n]$)

$vex[n][i]$: decoded VXC excitation signal at time index n ($0 \le i < FRM$)

$param[n]$: parameter at time index n

and interpolated parameters as:

$mdf\_pch[m]$: pitch lag value at time index m

$mdf\_vuv[m]$: V/UV index at time index m

$mdf\_lsp[m][i]$: decoded LSPs at time index m ($0 \le i < 10$)

$mdf\_send[m]$: number of harmonics at time index m

$mdf\_am[m][i]$: harmonic magnitudes at time index m ($0 \le i < send[n]$)

$mdf\_vex[m][i]$: decoded VXC excitation signal at time index m ($0 \le i < FRM$)

$mdf\_param[m]$: parameter at time index m

where n and m are the time indices (frame number) before and after the time scale modification. The frame intervals are both 20 ms.

spd : the ratio of speed change ($0.5 \le spd \le 2.0$)

N_1 : the duration of the original speech (total frame number)

N_2 : the duration of the speed controlled speech (total frame number)

fr_0, fr_1 : frame indices adjacent to the interpolation point

$left, right$: interpolation ratio

tmp_vuv : temporal V/UV index

2.5.5.3 Speed control process

Let us define the ratio of speed change as spd :

$$spd = N_1 / N_2 \tag{2.5.5.1}$$

where N_1 is the duration of the original speech and N_2 is the duration of the speed controlled speech. Therefore,

$0 \le n < N_1$ and $0 \le m < N_2$.

If the speed is controlled by the time scaling factor in the **speed** field of the AudioSource BIFS node, the speed change ratio is:

$$spd = 1 / speed$$

Basically, the time scale modified parameters are expressed as:

$$mdf\_param[m] = param[m \times spd] \tag{2.5.5.2}$$

where $param$ are: pch, vuv, lsp and am . In general, however, $m \times spd$ is not an integer.

We therefore define:

$$\begin{cases} fr_0 = \lfloor m \times spd \rfloor \\ fr_1 = fr_0 + 1 \end{cases} \tag{2.5.5.3}$$

to generate parameters at a time index $m \times spd$ by linearly interpolating the parameters at time indices fr_0 and fr_1. In order to execute linear interpolation, let us define:

$$\begin{cases} left = m \times spd - fr_0 \\ right = fr_1 - m \times spd \end{cases} \tag{2.5.5.4}$$

Then Eq.(2.5.5.2) can be approximated as:

$$mdf\_param[m] = param[fr_0] \times right + param[fr_1] \times left \tag{2.5.5.5}$$

where $param$ are: pch, vuv, lsp and am .

For $lsp[n][i]$ and $am[n][i]$, this linear interpolation is applied with the index i being fixed.

As for the parameter vex :

$vex[n][i]$ has excitation signals for UV frames by codebook look-up.

FRM samples from $vex[n][i]$ centering around the time $m \times spd$ are taken and the energy over the FRM samples are computed. Gaussian noise consisting of FRM samples is then generated and its norm is adjusted so that its

energy be equal to that of FRM samples taken from $vex[n][i]$. This gain adjusted Gaussian noise sequence is used for $mdf\_vex[m][i]$.

Principal operation of time scale modification can be expressed by the Eq.(2.5.5.5), however, the V/UV decisions at fr_0 and $fr1$, prior to the interpolation, have to be considered.

The interpolation and replacement strategies adapted to V/UV decisions are described below. In the explanation below, full voiced and mixed voiced ($vuv[n] \neq 0$) are grouped as "Voiced", and only $vuv[n] = 0$ case is regarded as "Unvoiced". In the case of variable rate coding, "Background Noise" mode ($vuv[n] = 1$) is also treated as "Unvoiced".

- When V/UV decisions at fr_0 and fr_1 are Voiced - Voiced:

  New V/UV index $mdf\_vuv[m]$ is obtained as follows;

  $tmp\_vuv = vuv[fr_0] \times right + vuv[fr_1] \times left$

  if (tmp_vuv > 2)

  $mdf\_vuv[m]$ = 3

  else if (tmp_vuv > 1)

  $mdf\_vuv[m]$ = 2

  else if (tmp_vuv > 0)

  $mdf\_vuv[m]$ = 1

  New pitch lag value $mdf\_pch[m]$ is obtained as follows;

  if (0.57 < $pch[fr_0] / pch[fr_1]$ && $pch[fr_0] / pch[fr_1]$ < 1.75)

  $mdf\_pch[m] = pch[fr_0] \times right + pch[fr_1] \times left$

  else

  if ($left$ < $right$)

  $mdf\_pch[m] = pch[fr_0]$

  else

  $mdf\_pch[m] = pch[fr_1]$

  All the other parameters are interpolated by Eq.(2.5.5.5).

- When V/UV decisions at fr_0 and fr_1 are Unvoiced - Unvoiced:
  All the parameters are interpolated by Eq.(2.5.5.5) except for mdf_vex. $mdf\_vex[m][i]$ is generated by Gaussian noise having the same energy as that of FRM samples taken from $vex[n][i]$ centering around the time $m \times spd$.

- When V/UV decisions at fr_0 and fr_1 are Voiced - Unvoiced:
  If $left < right$
  All the parameters at time fr_0 are used instead of computing the parameters at $m \times spd$.
  If $left \geq right$
  All the parameters at time fr_1 are used instead of computing the parameters at $m \times spd$.
  $mdf\_vex[m][i]$ is also generated by Gaussian noise having the same energy as that of FRM samples from $vex[fr_1][i]$. $(0 \leq i < FRM)$

- When V/UV decisions at fr_0 and fr_1 are Unvoiced - Voiced:
  If $left < right$
  All the parameters at time fr_0 are used instead of computing the parameters at $m \times spd$.
  $mdf\_vex[m][i]$ is also generated by Gaussian noise having the same energy as that of FRM samples from $vex[fr0][i]$. $(0 \leq i < FRM)$
  If $left \geq right$
  All the parameters at time fr_1 are used instead of computing the parameters at $m \times spd$.

In this manner, one obtain all the necessary parameters for the HVXC decoder. Just by applying these modified parameters, $mdf\_param[m]$, to “Voiced Component Synthesizer” and “Unvoiced Component Synthesizer” in the same way as usual (normal) decoding process, one obtain the time scale modified output.

Apparently, when $N_2 < N_1$, speed up decoding is executed, and when $N_2 > N_1$ speed down decoding is executed. Power spectrum and pitch are not affected by this speed control, thus we can obtain good quality speech for the speed control factor of about $0.5 < spd < 2.0$.

2.5.6 Voiced component synthesizer

2.5.6.1 Tool description

The voiced component synthesizer consists of the steps below;

- harmonic magnitudes modification
- harmonic excitation synthesis
- noise component addition
- LPC synthesis
- postfilter

An efficient harmonic excitation synthesis method is first used to obtain a periodic excitation waveform from harmonic magnitudes. By adding a noise component to the periodic waveform the voiced excitation signal is obtained, which is then fed into the LPC synthesis filter and postfilter to generate a voiced speech signal. The configuration of the postfilter is not normative and described in Annex 2.B.

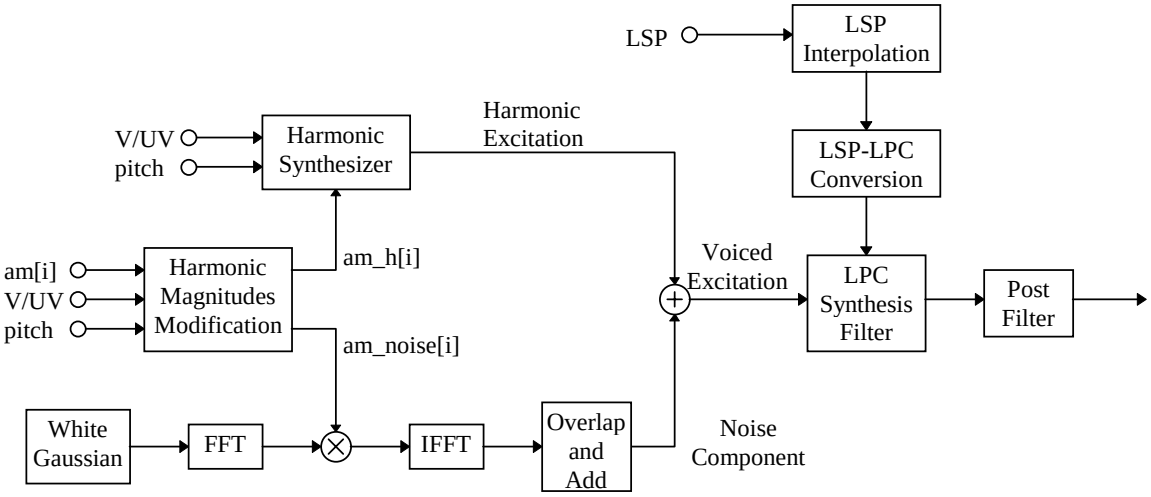


Figure 2.8 — Voiced component synthesizer

2.5.6.2 Definitions

Definitions of constants

PI : $\pi = 3.14159265358979\ldots$

FRM : frame interval (=160)

$SAMPLE$: frame length of analysis (=256)

$WAVE\_LEN$: length of one pitch period waveform (=128)

$WDEVI$: threshold of the ratio of the fundamental frequency change (=0.1)

LD_LEN : decode interval shifting for low delay mode (=20)

RND_MAX : maximum number generated by random number generator (=0x7fffffff)

$SCALEFAC$: scale factor for the modified harmonic magnitudes for noise component generation (=10)

$c\_dis[i]$: trapezoid window for the harmonic synthesis of pitch discontinuous waveform shown in the Figure 2.9 ($0 \le i < FRM + LD\_LEN$)

$ham[i]$: Hamming window ($0 \le i < SAMPLE$)

$ham\_z[i]$: zero padded Hamming window ($0 \le i < 2 \times FRM$)

$HAMLD$: the length of the Hamming window for low delay mode (=240)

P : LPC order (=10)

Definitions of variables

$am2[i]$: harmonic magnitudes at the ending boundary of decode interval. They are obtained as $am[i]$ in the subclause 2.5.3.3.

$am\_h[i]$: modified harmonic magnitudes at the ending boundary of the decode interval

$am\_noise[i]$: modified harmonic magnitudes for noise component generation at the ending boundary of the decode interval

$vuv2$: V/UV index at the ending boundary of the decode interval. It is obtained as **VUV** from the bitstream.

$vuv1$: V/UV index at the beginning boundary of the decode interval

$vuv0$: V/UV index at the beginning boundary of the previous decode interval

$pch2$: pitch lag value[sample] at the ending boundary of the decode interval. It is obtained as pch (pitch lag value in the current frame) in the subclause 2.5.3.3.

$pch1$: pitch lag value[sample] at the beginning boundary of the decode interval

$send2$: the number of harmonics at the ending boundary of the decode interval

$send1$: the number of harmonics at the beginning boundary of the decode interval

$w02$: the fundamental frequency at the ending boundary of the decode interval [rad/sample]

$w01$: the fundamental frequency at the beginning boundary of the decode interval [rad/sample]

$pha2[i]$: harmonic phase values at the ending boundary of the decode interval

$pha1[i]$: harmonic phase values at the beginning boundary of the decode interval

$ovsr2$: over sampling ratio at the ending boundary of the decode interval

$ovsr1$: over sampling ratio at the beginning boundary of the decode interval

$ovsrc$: linearly interpolated over sampling ratio

$wave2[i]$: one pitch period waveform generated from pitch and harmonic magnitudes at the ending boundary of the decode interval $(0 \le i < WAVE\_LEN)$

$wave1[i]$: one pitch period waveform generated from pitch and harmonic magnitudes at the beginning boundary of the decode interval $(0 \le i < WAVE\_LEN)$

$lp12,\ lp12r$: the length of over-sampled waveform needed to reconstruct a waveform of the decode interval (in case of continuous pitch transition)

$lp1,\ lp2,\ lp2r$: the length of over-sampled waveform needed to reconstruct a waveform of the decode interval (in case of discontinuous pitch transition)

st : offset for cyclic extension of $wave2[i]$ at the beginning boundary of the decode interval

$iflat1, iflat2$: the length of the period where no parameter interpolation is done in over sampled domain (for low delay mode)

$out2[]$: cyclically extended waveform generated from $wave2[i]$

$out1[]$: cyclically extended waveform generated from $wave1[i]$

$out3[]$: weighted overlapped and added cyclically extended waveform within the decode interval generated from $out1[]$ and $out2[]$ (in case of continuous pitch transition)

$sv2[i]$: re-sampled waveform from $out2[]$ (in case of discontinuous pitch transition, $0 \le i < FRM$)

$sv1[i]$: re-sampled waveform from $out1[]$ (in case of discontinuous pitch transition, $0 \le i < FRM$)

$sv[i]$: generated voiced excitation signal ($0 \le i < FRM$)

$ns[i]$: Gaussian noise with zero mean and unit variance ($0 \le i < SAMPLE$)

$wns[i]$: Hamming windowed Gaussian noise ($0 \le i < SAMPLE$)

$rms[i]$: spectral amplitude array ($0 \le i \le SAMPLE / 2$)

$ang[i]$: spectral phase array ($0 \le i \le SAMPLE / 2$)

$re[i]$: real part of the FFT coefficients ($0 \le i < SAMPLE$)

$im[i]$: imaginary part of the FFT coefficients ($0 \le i < SAMPLE$)

$w0$: fundamental frequency of the current frame, where 2π is expressed as $SAMPLE(=256)$

$cns[i]$: the IFFT result of the current frame for noise component generation ($0 \le i < SAMPLE$)

$cns\_z[i]$: zero padded array to $cns[]$ ($0 \le i < 2 \times FRM$)

$cns\_z\_p[i]$: the previous frame's $cns\_z[]$ for overlap and add ($0 \le i < 2 \times FRM$)

$add\_uv[i]$: generated noise component at the decode interval ($0 \le i < FRM$)

$lsp2[]$: the de-quantized LSPs of the current frame obtained as $qLsp[]$ in the subclause 2.5.2.3.

$lsp1[]$: the de-quantized LSPs of the previous frame

$lspip[][]$: the interpolated LSPs

$alpaip[][]$: the linear predictive coefficients converted from the interpolated LSPs $lspip[][]$

Definitions of functions

$random()$: random number generator that returns random numbers ranging from 0 to RND_MAX

$ceil(x)$: a function that returns the least integer greater than or equal to x

$floor(x)$: a function that returns the greatest integer less than or equal to x

2.5.6.3 Synthesis process

Voiced component signal is synthesized as shown below. The synthesis algorithm can be applied for both normal delay mode and low delay mode. For low delay mode, decode interval shifting, LD_LEN , is used. The synthesized waveform covers from $N = -160 + LD\_LEN$ [sample] to $N = 0 + LD\_LEN$ [sample]. $N = 0$ represents the center of the current frame. If the frame shifting is 0 ($LD\_LEN = 0$), the synthesized waveform is identical to that of the normal delay mode. This is shown in the figure below.

During the period from $N = 0$ to $N = LD\_LEN$, pitch lag, harmonic magnitudes and LSP parameters are not interpolated and hold. If $LD\_LEN = 0$, decoder delay is 10[ms] and if $LD\_LEN = 20$, decoder delay is 7.5[ms].

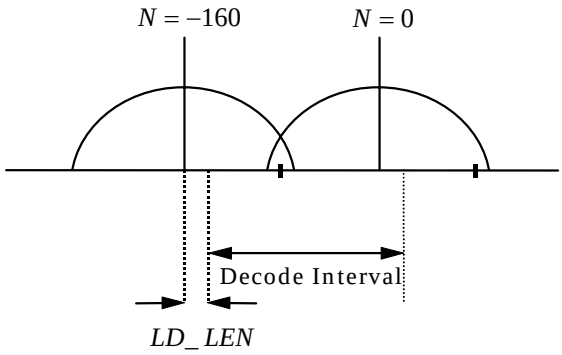


Figure 2.9 — Decode interval

2.5.6.3.1 Harmonic magnitudes modification

Harmonic magnitudes, $am2[i]$, are modified for harmonic excitation synthesis and noise component generation independently according to the V/UV index of the current frame, $vuv2$.

The modified harmonic magnitudes $am\_h[i]$ for harmonic excitation synthesis and $am\_noise[i]$ for noise component generation are obtained as described below.

When V/UV index, $vuv2$, is 0:

Do Nothing (just VXC decoder works).

When V/UV index, $vuv2$, is 1:

$$am\_h[i] = \begin{cases} am2[i] & (1 \le i < send2 \times B\_TH1) \\ am2[i] \times AH1 & (send2 \times B\_TH1 \le i \le send2) \end{cases}$$

$$am\_noise[i] = \begin{cases} 0 & (1 \le i < send2 \times B\_TH1) \\ SCALEFAC \times am2[i] \times AN1 & (send2 \times B\_TH1 \le i \le send2) \end{cases}$$

When V/UV index, $vuv2$, is 2:

$$am\_h[i] = \begin{cases} am2[i] & (1 \le i < send2 \times B\_TH2) \\ am2[i] \times AH2 & (send2 \times B\_TH2 \le i < send2 \times B\_TH2\_2) \\ am2[i] \times AH2\_2 & (send2 \times B\_TH2\_2 \le i \le send2) \end{cases}$$

$$am\_noise[i] = \begin{cases} 0 & (1 \le i < send2 \times B\_TH2) \\ SCALEFAC \times am2[i] \times AN2 & (send2 \times B\_TH2 \le i < send2 \times B\_TH2\_2) \\ SCALEFAC \times am2[i] \times AN2\_2 & (send2 \times B\_TH2\_2 \le i \le send2) \end{cases}$$

When V/UV index, $vuv2$, is 3:

$$am\_h[i] = \begin{cases} am2[i] & (1 \le i < send2 \times B\_TH3) \\ am2[i] \times AH3 & (send2 \times B\_TH3 \le i \le send2) \end{cases}$$

$$am\_noise[i] = \begin{cases} 0 & (1 \le i < send2 \times B\_TH3) \\ SCALEFAC \times am2[i] \times AN3 & (send2 \times B\_TH3 \le i \le send2) \end{cases}$$

Table 2.63 — Constant values for harmonic magnitude modification

2 kbps:

B_TH1	AN1	AH1	B_TH2	AN2	AH2	B_TH2_2	AN2_2	AH2_2	B_TH3	AN3	AH3
0.5	0.4	0.8	0.5	0.3	0.9	0.85	0.5	0.5	0.7	0.2	1.0

4 kbps:

B_TH1	AN1	AH1	B_TH2	AN2	AH2	B_TH2_2	AN2_2	AH2_2	B_TH3	AN3	AH3
0.5	0.3	0.9	0.5	0.3	0.9	0.85	0.5	0.5	0.8	0.2	1.0

2.5.6.3.2 Harmonic excitation synthesis

Harmonic excitation signal $sv[i]\,(0 \le i < FRM)$ can be obtained by a fast synthesis method composed of an IFFT and sampling rate conversion.

First, using harmonic magnitudes and phase values, a waveform over one pitch period is generated by an IFFT. According to the pitch continuity, one of two different operations of cyclic extension and re-sampling of a waveform over one pitch period is selected and performed to obtain the harmonic excitation signal.

Generation of a waveform over one pitch period

The fundamental frequency at the beginning boundary of the decode interval, $w01$, is computed as

```
w01 = 2.0*PI/pch1;
```

The fundamental frequency at the ending boundary of the decode interval, $w02$, is computed as

```
w02 = 2.0*PI/pch2;
```

Harmonic phase values at the ending boundary of the decode interval, $pha2[]$,are computed from the harmonic phase values at the beginning boundary, $pha1[]$. When both of the V/UV index at the beginning of the current decode interval, $vuv1$, and the V/UV index at the beginning boundary of the previous decode interval, $vuv0$, are 0 (Unvoiced), harmonic phase values $pha2[]$ are initialized using random phase values uniformly distributing between 0 to 0.5π.

For nomal delay mode :

```
if (vuv1 == 0 && vuv0 == 0) {
   for (i = 0; i < WAVE_LEN/2; i++)
      pha2[i] = 0.5*PI*(float)random()/(float)RND_MAX;
}
else {
   for (i = 0; i < WAVE_LEN/2; i++)
      pha2[i] = pha1[i] + 0.5*(w01+w02)*i*FRM;
}
```

For low delay mode:

```
if (vuv1 == 0 && vuv0 == 0) {
   for (i = 0; i < WAVE_LEN/2; i++)
      pha2[i] = 0.5*PI*(float)random()/(float)RND_MAX;
}
else {
   for (i = 0; i < WAVE_LEN/2; i++) {
      pha2[i] = pha1[i] + 0.5*(w01+w02)*i*(FRM-LD_LEN) + w01*i*LD_LEN;
}
```

At the end boundary of the decode interval we have a spectrum with $send2$ harmonics, whose magnitudes are $am\_h[i]\,(1 \le i \le send2)$ and phase values are $pha2[i]\,(1 \le i \le send2)$. Appending zeros to these arrays yields new arrays with $WAVE\_LEN/2(=64)$ components ranging from 0 to π. If $send2$ is greater than 63, first 64 values of $am\_h[i]$ and $pha2[i]$ are used. 128 point IFFT is applied to these arrays of magnitudes and phase values with the constraint that the results be real numbers. Now we have an over-sampled waveform over one pitch period, $wave2[i]\,(0 \le i < WAVE\_LEN)$.

$WAVE\_LEN(=128)$ samples are used to express the one pitch period of the waveform. Since the actual pitch lag value is $pch2$, the over-sampling rate $ovsr2$ is

```
ovsr2 = WAVE_LEN/pch2;
```

Similarly the over-sampling rate $ovsr1$ for the waveform over one pitch period at the beginning boundary of the decode interval, $wave1[]$,is

```
ovsr1 = WAVE_LEN/pch1;
```

Pitch continuity check

When the ratio of the fundamental frequencies, $|(w02 - w01) / w02|$, is less than $WDEVI(= 0.1)$, it is regarded that pitch transition is continuous. In such a case, the fundamental frequencies and harmonic magnitudes are linearly interpolated between the beginning and the ending of the decode interval. Otherwise it is regarded that pitch transition is discontinuous and the fundamental frequencies and harmonic magnitudes are not linearly interpolated. In this case, independently synthesized periodic waveforms are added using appropriate weighting functions.

Cyclic extension and re-sampling operation (continuous pitch transition)

Waveforms over one pitch period, $wave1[]$ and $wave2[]$, are cyclically extended respectively to have sufficient length in the over over-sampled domain.

The length of over-sampled waveform needed to reconstruct a waveform of length $FRM(= 160)$ at the original sampling rate (8 kHz) is at most $lp12$.

For normal delay mode:

```
lp12 = ceil(FRM*0.5*(over1+ovsr2));
lp12r = floor(FRM*0.5*(ovsr1+ovsr2)+0.5);
st = WAVE_LEN - (lp12r%WAVE_LEN);
for (i = 0; i < lp12; i++) {
   out1[i] = wave1[i%WAVE_LEN];
   out2[i] = wave2[(st+i)%WAVE_LEN];
}
```

For low delay mode:

```
lp12 = ceil((FRM-LD_LEN)*0.5*(over1+ovsr2)+LD_LEN*ovsr1);
lp12r = floor((FRM-LD_LEN)*0.5*(ovsr1+ovsr2)+LD_LEN*ovsr1+0.5);
st = WAVE_LEN - (lp12r%WAVE_LEN);
iflat1 = floor(ovsr1*LD_LEN+0.5);
iflat2 = floor(ovsr2*LD_LEN+0.5);
for (i = 0; i < lp12+iflat2+10; i++) {
   out1[i] = wave1[i%WAVE_LEN];
   out2[i] = wave2[(st+i)%WAVE_LEN];
}
```

These two waveforms, $out1[]$ and $out2[]$, have the same “pseudo” pitch period (=$WAVE\_LEN$ [sample]) and they are aligned. So simply adding these two waveforms using triangular windows produces the waveform $out3[]$.

For normal delay mode:

```
for (i = 0; i < lp12; i++)
   out3[i] = out1[i]*(float)(lp12-i)/(float)lp12 + out2[i]*(float)i/(float)lp12;
```

For low delay mode:

```
for (i = 0; i < iflat1; i++)
   out3[i] = out1[i];

for (i = iflat1; i < lp12; i++)
   out3[i] = out1[i]*(float)(lp12-i)/(float)(lp12-iflat1)
             + out2[i]*(float)(i-flat1)/(float)(lp12-iflat1);

for (i = lp12; i < lp12+iflat2; i++)
     out3[i] = out2[i];
```

Finally, $out3[]$ has to be re-sampled so that the resulting waveform can be expressed at the original sampling rate (8 kHz). This operation brings the waveform back from the "pseudo" pitch domain to the actual pitch domain. In principle, the re-sampling operation is just

$$sv[i] = out3[f(i)] \quad (0 \le i < FRM)$$

where

$$f(i) = \int_0^i \left(ovsr1 \times \frac{FRM - t}{FRM} + ovsr2 \times \frac{t}{FRM} \right) dt .$$

The function $f(i)$ maps the time index i of the original sampling rate (8 kHz) to the time index at the over-sampled rate under the condition that the fundamental frequencies, $w01$ and $w02$, is linearly interpolated. Since $f(i)$ does not return an integer value, $sv[i]$ is obtained by linearly interpolating $out3[\lfloor f(i) \rfloor]$ and $out3[\lceil f(i) \rceil]$.

For normal delay mode:

```
sv[0] = out3[0];
ffi = 0;
for (i = 1; i < FRM; i++) {
   ovsrc = ovsr1*(float)(FRM-i)/(float)FRM + over2*(float)i/(float)FRM;
   ffi += ovsrc;
   ffim = floor(ffi);
   ffip = ceil(ffi);
   if (ffim == ffip)
      sv[i] = out3[(int)ffip];
   else
      sv[i] = (ffi-ffim)*out3[(int)ffip] + (ffip-ffi)*out3[(int)ffim];
}
```

For low delay mode:

```
sv[0] = out3[iflat1];
ffi = 0;
for (i = 1; i < FRM; i++) {
   ovsrc = ovsr1*(float)(FRM-LD_LEN-i)/(float)(FRM-LD_LEN)
           + over2*(float)i/(float)(FRM-LD_LEN);
   ffi += ovsrc;
   ffim = floor(ffi);
   ffip = ceil(ffi);
   if (ffim == ffip)
      sv[i] = out3[(int)ffip+iflat1];
   else
      sv[i] = (ffi-ffim)*out3[(int)ffip+iflat1]
                + (ffip-ffi)*out3[(int)ffim+iflat1];
}
```

Cyclic extension and re-sampling operation (discontinuous pitch transition)

Waveforms over one pitch period, $wave1[]$ and $wave2[]$ are cyclically extended to have sufficient length in the over-sampled domain. At both the beginning and ending boundary of the decode interval, cyclically extended waveforms, $out1[]$ and $out2[]$, are obtained.

```
lp1 = ceil(FRM*ovsr1);
lp2 = ceil(FRM*ovsr2);
lp2r = floor(FRM*ovsr2+0.5);
st = WAVE_LEN - (lp2r%WAVE_LEN);
```

For normal delay mode:

```
for (i = 0; i < lp1; i++)
   out1[i] = wave1[i%WAVE_LEN];
for (i = 0; i < lp2; i++)
   out2[i] = wave2[(st+i)%WAVE_LEN];
```

For low delay mode:

```
iflat1 = floor(ovsr1*LD_LEN+0.5);
iflat2 = floor(ovsr2*LD_LEN+0.5);

for (i = 0; i < lp1+iflat1+10; i++)
   out1[i] = wave1[i%WAVE_LEN];
for (i = 0; i < lp2+iflat2+10; i++)
   out2[i] = wave2[(st+i)%WAVE_LEN];
```

where $lp1$ and $lp2$ are the length of over-sampled waveforms needed to reconstruct a waveform of length $FRM(=160)$ at the original sampling rate (8 kHz).

These two waveforms, $out1[]$ and $out2[]$, are re-sampled independently using the same linear interpolation method as in the case of "continuous pitch transition".

For normal delay mode:

```
sv1[0] = out1[0];
ffi = 0;
for (i = 1; i < FRM; i++) {
   ffi += ovsr1;
   ffim = floor(ffi);
   ffip = ceil(ffi);
   if (ffim == ffip)
      sv1[i] = out1[(int)ffip];
   else
      sv1[i] = (ffi-ffim)*out1[(int)ffip] + (ffip-ffi)*out1[(int)ffim];
}
sv2[0] = out2[0];
ffi = 0;
for (i = 1; i < FRM; i++) {
   ffi += ovsr2;
   ffim = floor(ffi);
   ffip = ceil(ffi);
   if (ffim == ffip)
      sv2[i] = out2[(int)ffip];
   else
      sv2[i] = (ffi-ffim)*out2[(int)ffip] + (ffip-ffi)*out2[(int)ffim];
}
```

For low delay mode:

```
sv1[0] = out1[iflat1];
ffi = 0;
for (i = 1; i < FRM; i++) {
   ffi += ovsr1;
   ffim = floor(ffi);
   ffip = ceil(ffi);
   if (ffim == ffip)
      sv1[i] = out1[(int)ffip+iflat1];
   else
      sv1[i] = (ffi-ffim)*out1[(int)ffip+iflat1]
                + (ffip-ffi)*out1[(int)ffim+iflat1];
}
sv2[0] = out2[iflat2];
ffi = 0;
```

```
for (i = 1; i < FRM; i++) {
   ffi += ovsr2;
   ffim = floor(ffi);
   ffip = ceil(ffi);
   if (ffim == ffip)
      sv2[i] = out2[(int)ffip+iflat2];
   else
      sv2[i] = (ffi-ffim)*out2[(int)ffip+iflat2]
               + (ffip-ffi)*out2[(int)ffim+iflat2];
}
```

Using re-sampled waveforms at the original sampling rate (8 kHz), $sv1[]$ and $sv2[]$, are then overlapped and added with the trapezoid window $c\_dis[]$ shown in the Figure 2.10, where $HM\_UP = HM\_DOWN = 60, HM\_FLAT = 50$.

```
for (i = 0; i < HM_FLAT; i++)
   c_dis[i] = 1.0f;
for (i = HM_FLAT; i < HM_FLAT+HM_DOWN; i++)
   c_dis[i] = (-i+(HM_FLAT+HM_DOWN))/(float)HM_DOWN;
for (i = HM_FLAT+HM_DOWN; i < FRM; i++)
   c_dis[i] = 0.0f;
```

For normal delay mode:

```
for (i = 0; i < FRM; i++)
   sv[i] = sv1[i]*c_dis[i] + sv2[i]*(1.0-c_dis[i]);
```

For low delay mode:

```
for (i = 0; i < FRM; i++)
   sv[i] = sv1[i]*c_dis[i+LD_LEN] + sv2[i]*(1.0-c_dis[i+LD_LEN]);
}
```

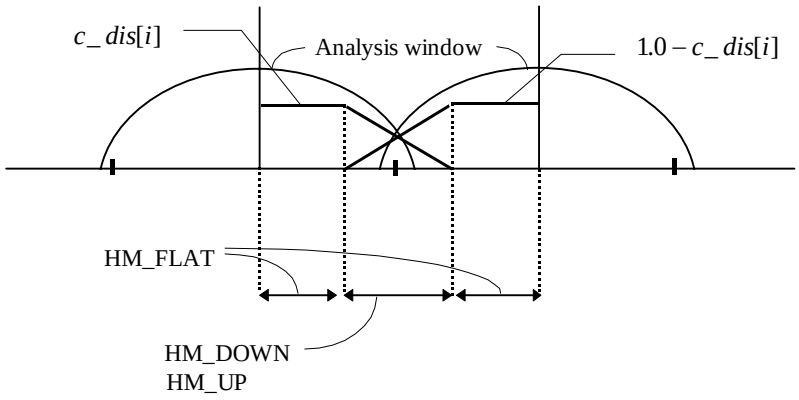


Figure 2.10 — Synthesis window for discontinuous pitch

2.5.6.3.3 Noise component generation

For the noise component generation for voiced excitation, white Gaussian noise is first generated. It is then colored and gain adjusted by the modified harmonic magnitudes, $am\_noise[]$, and weighted overlap and add is used to generate a continuous noise component in the time domain.

Hamming window of length $SAMPLE(=256)$, $ham[i]$, is first defined.

For normal delay mode,

```
   for (i = 0; i < SAMPLE; i++)
     ham[i] = 0.54-0.46*cos(2.0*PI*i/(SAMPLE-1));
```

For low delay mode,

```
   for (i = 0; i < (SAMPLE-HAMLD)/2; i++)
      ham[i] = 0.0;
   for (i = (SAMPLE-HAMLD)/2; i < (SAMPLE+HAMLD)/2; i++)
      ham[i] = 0.54-0.46*cos(2.0*PI*i/(SAMPLE-1));
   for (i = (SAMPLE+HAMLD)/2; i < SAMPLE; i++)
      ham[i] = 0.0;
```

Window $ham[]$ is normalized to have a unit energy.

Let $ns[i]\,(0 \le i < SAMPLE)$ be samples of white Gaussian noise with zero mean and unit variance. Hamming window $ham[i]$ is then multiplied to $ns[i]$ and $wns[i]$ is obtained.

```
   for (i = 0; i < SAMPLE; i++)
      wns[i] = ns[i]*ham[i];
```

256 point FFT of $wns[i]$ is computed and spectral amplitude array $rms[i]\,(0 \le i \le SAMPLE/2)$ and spectral phase array $ang[i]\,(0 \le i \le SAMPLE/2)$ are computed as

```
   for (i = 0; i <= SAMPLE/2; i++) {
      rms[i] = sqrt(re[i]*re[i]+im[i]*im[i]);
      ang[i] = atan2(im[i], re[i]);
   }
```

where $re[i]\,(0 \le i < SAMPLE)$ and $im[i]\,(0 \le i < SAMPLE)$ are real part and imaginary part of the FFT coefficients respectively. Then spectral amplitude $rms[]$ is colored and gain adjusted by the modified harmonic magnitudes, $am\_noise[]$. ($w0$ is the fundamental frequency of the current frame where $w0$ of value $SAMPLE$ represents 2π.)

```
   for (i = 0; i <= send2; i++) {
      if (i == 0)
         lb = 0;
      else
         lb = ub+1;
      if (i == send2)
         ub = SAMPLE/2
      else
         ub = floor((float)i*w0+w0/2.0+0.5);
      if (ub >= SAMPLE/2)
         ub = SAMPLE/2;
      bw = ub-lb+1;
      s = 0.0;
      for (j = lb; j <= ub; j++)
         s += rms[j]*rms[j];
      s = sqrt(s/(float)bw);
      for (j = lb; j <= ub; j++)
         rms[j] *= am_noise[i]/s;
   }
```

256 point IFFT is computed with the colored and gain adjusted spectral amplitude array $rms[]$ and the original spectral phase array $ang[]$ with a constraint that the result be real numbers. Let the result of IFFT be $cns[i]\,(0 \le i < SAMPLE)$.

It should be noted that when the current frame is “Unvoiced”

frame ($vuv2 = 0$), $cns[i] = 0.0\,(0 \le i < SAMPLE)$.

In order to generate noise component signal over the decode interval, weighted overlap and add with the previous frame's IFFT result is performed. The array $cns\_z[i]\,(0 \le i < 2 \times FRM)$ shown below is obtained by padding ($FRM - SAMPLE/2$) of zeros to both sides (beginning and ending) of $cns[]$.

```
for (i = 0; i < FRM-SAMPLE/2; i++)
   cns_z[i] = 0.0;
for (i = FRM-SAMPLE/2; i < FRM+SAMPLE/2; i++)
   cns_z[i] = cns[i-FRM+SAMPLE/2];
for (i = FRM+SAMPLE/2; i < 2*FRM; i++)
   cns_z[i] = 0.0;
```

In the same manner, zero padded Hamming window array $ham\_z[i]\,(0 \le i < 2 \times FRM)$ is defined as:

```
for (i = 0; i < FRM-SAMPLE/2; i++)
   ham_z[i] = 0.0;
for (i = FRM-SAMPLE/2; i < FRM+SAMPLE/2; i++)
   ham_z[i] = ham[i-FRM+SAMPLE/2];
for (i = FRM+SAMPLE/2; i < 2*FRM; i++)
   ham_z[i] = 0.0;
```

Let us denote the $cns\_z[]$ of the previous frame $cns\_z\_p[]$. Now noise component over the decode interval, $add\_uv[i]\,(0 \le i < FRM)$, is obtained by combining $cns\_z[]$ and $cns\_z\_p[]$.

For normal delay mode,

```
for (i = 0; i < FRM; i++)
   add_uv[i] = (cns_z_p[FRM+i]*ham_z[FRM+i]+cns_z[i]*ham_z[i])
             /(ham_z[FRM+i]*ham_z[FRM+i]+ham_z[i]*ham_z[i])
```

For low delay mode, noise component at the shifted decode interval is obtained as follows:

```
for (i = 0; i < FRM-LD_LEN; i++)
   add_uv[i] = (cns_z_p[FRM+i+LD_LEN]*ham_z[FRM+i+LD_LEN]+
                cns_z[i+LD_LEN]*ham_z[i+LD_LEN])
                /(ham_z[FRM+i+LD_LEN]*ham_z[FRM+i+LD_LEN]
                   +ham_z[i+LD_LEN]*ham_z[i+LD_LEN]);
for (i = FRM-LD_LEN; i < FRM; i++)
   add_uv[i] = cns_z[i+LD_LEN]/ham_z[i+LD_LEN]);
```

Noise component $add\_uv[]$ is added to harmonic excitation signal $sv[]$ to make a voiced excitation signal:

```
for (i = 0; i < FRM; i++)
   sv[i] += add_uv[i];
```

2.5.6.3.4 LPC synthesis

The voiced excitation signal obtained above, $sv[i]\,(0 \le i < FRM)$, is fed into the LPC synthesis filter, whose coefficients are updated every 2.5 ms (=20 samples).

By linearly interpolating de-quantized LSPs of the previous and the current frame, $lsp1[][]$ and $lsp2[][]$, the 8 sets of the interpolated LSPs, $lspip[][]$, are obtained.

```
for (i = 0; i < 8; i++)
   for (j = 0; j < P; j++)
      lspip[i][j] = (2.0*i+1.0)/16.0*lsp2[j] + (16.0-2.0*i-1.0)/16.0*lsp1[j];
```

For the low delay mode, since the decode interval is shifted by $LD\_LEN(=20)$ samples (2.5 ms), the LSPs interpolation is carried out for the first 17.5 ms of the decode interval shown in Figure 2.9, and the LSPs of the current frame, $lsp2[]$, is used for the last 2.5ms without interpolation.

```
for (i = 0; i < 7; i++)
   for (j = 0; j < P; j++)
      lspip[i][j] = (2.0*i+1.0)/14.0*lsp2[j] + (14.0-2.0*i-1.0)/14.0*lsp1[j];
for (j = 0; j < P; j++)
   lspip[7][j] = lsp2[j];
```

The 8 sets of the interpolated LSPs, $lspip[][]$, are converted to the linear predictive coefficients, $alpaip[][]$, respectively.

The transfer function of the i -th interval of 2.5 ms (20 samples) of the LPC synthesis filter is

$$H_i(z) = \frac{1}{\sum_{n=0}^{P} alphaip[i][n]\, z^{-n}} \quad (0 \le i < 8)$$

Output of the LPC synthesis filter is fed into the postfilter described in subclause 2.B.1.3.1. The output signal is in the range from -32768 to 32767.

2.5.7 Unvoiced component synthesizer

2.5.7.1 Tool description

The unvoiced component synthesizer is composed of three steps, which are windowing the unvoiced excitation signal, LPC synthesis filter, and post filter operation. For unvoiced segments, the VXC (CELP) scheme is used.

2.5.7.2 Definitions

Definitions of constants

FRM : frame interval (=160)

LD_LEN : decode interval shifting for low delay mode (=20)

$w\_celp\_up[i]$: the window from voiced frame to unvoiced frame ($0 \le i < FRM + LD\_LEN$)

$w\_celp\_down[i]$: the window from unvoiced frame to voiced frame ($0 \le i < FRM + LD\_LEN$)

P : LPC order (=10)

Definitions of variables

$qRes[i]$: decoded unvoiced excitaion signal obtained as $res[i]$ in subclause 2.5.4.3 ($0 \le i < FRM$)

$old\_qRes[i]$: the last half of the decoded unvoiced excitation signal of the previous frame ($0 \le i < FRM/2$)

$suv[i]$: unvoiced excitaion signal over the decode interval ($0 \le i < FRM$)

$vuv2$: V/UV index of the current frame obtained as **VUV** in a bitstream

$vuv1$: V/UV index of the previous frame

$lsp2[]$: de-quantized LSPs of the current frame obtained as $qLsp[]$ in the subclause 2.5.2.3

$lsp1[]$: de-quantized LSPs of the previous frame

$alpha2[]$: LPC coefficients converted from the LSPs $lsp2[]$

$alpha1[]$: LPC coefficients converted from the LSPs $lsp1[]$

2.5.7.3 Synthesis process

An unvoiced excitation signal at the decode interval, $suv[]$, is generated from the decoded unvoiced excitaion signal of the current frame, $qRes[]$, and the last half of the decoded unvoiced excitation signal of the previous frame, $old\_qRes[]$.

```
for (i = 0; i < FRM/2; i++) {
   suv[i] = old_qRes[i];
   suv[i+FRM/2] = qRes[i];
   old_qRes[i] = qRes[i+FRM/2];
}
```

For low delay mode, excitation signal of $LD\_LEN(=20)$ samples shifted version is used.

```
for (i = 0; i < FRM/2-LD_LEN; i++) {
   suv[i] = old_qRes[i+LD_LEN];
   suv[i+FRM/2] = qRes[i+LD_LEN];
   old_qRes[i] = qRes[i+FRM/2];
}
for (i = FRM/2-LD_LEN; i < FRM/2; i++) {
   suv[i] = qRes[i-FRM/2+LD_LEN];
   suv[i+FRM/2] = qRes[i+LD_LEN];
   old_qRes[i] = qRes[i+FRM/2];
}
```

The unvoiced excitation signal generated is windowed to be smoothly connected to a voiced frame. Figure 2.11 and Figure 2.12 shows the window shape for excitation waveform where V/UV is changed from unvoiced to voiced and from voiced to unvoiced respectively. The parameters in the figure are set as: $TD\_UP = 30$, $TD\_FLAT = 50$, $TD\_DOWN = 30$, $HM\_DOWN = 60$, $HM\_FLAT = 50$, $HM\_UP = 60$. These windows for an unvoiced frame are used only when an unvoiced frame is placed adjacent to voiced or mixed voiced frame.

```
for (i = 0; i < FRM-TD_UP-TD_FLAT; i++)
   w_celp_up[i] = 0.0;
for (i = FRM-TD_UP-TD_FLAT; i < FRM-TD_FLAT; i++)
   w_celp_up[i] = (float)(i-FRM+TD_UP+TD_FLAT)/(float)TD_UP;
for (FRM-TD_FLAT; i < FRM+LD_LEN; i++)
   w_celp_up[i] = 1.0;

for (i = 0; i < TD_FLAT; i++)
   w_celp_down[i] = 1.0;
for (i = TD_FLAT; i < TD_FLAT+TD_DOWN; i++)
   w_celp_down[i] = (float)(TD_FLAT+TD_DOWN-i)/(float)TD_DOWN;
for (i = TD_FLAT+TD_DOWN; i < FRM+LD_LEN; i++)
   w_celp_down[i] = 0.0;
```

For normal delay mode,

```
if (vuv1 != 0 && vuv2 != 0) {
   for (i = 0; i < FRM; i++)
      s_uv[i] = 0.0f;
}
else if (vuv1 != 0 && vuv2 == 0) {
   for (i = 0; i < FRM; i++)
      s_uv[i] *= w_celp_up[i];
}
else if (vuv1 == 0 && vuv2 != 0) {
   for (i = 0; i < FRM; i++)
      s_uv[i] *= w_celp_down[i]
}
```

For low delay mode, windowing position is shifted by LD_LEN samples.

```
if (vuv1 != 0 && vuv2 != 0) {
   for (i = 0; i < FRM; i++)
      s_uv[i] = 0.0f;
}
else if (vuv1 != 0 && vuv2 == 0) {
   for (i = 0; i < FRM; i++)
      s_uv[i] *= w_celp_up[i+LD_LEN];
}
else if (vuv1 == 0 && vuv2 != 0) {
   for (i = 0; i < FRM; i++)
      s_uv[i] *= w_celp_down[i+LD_LEN]
}
```

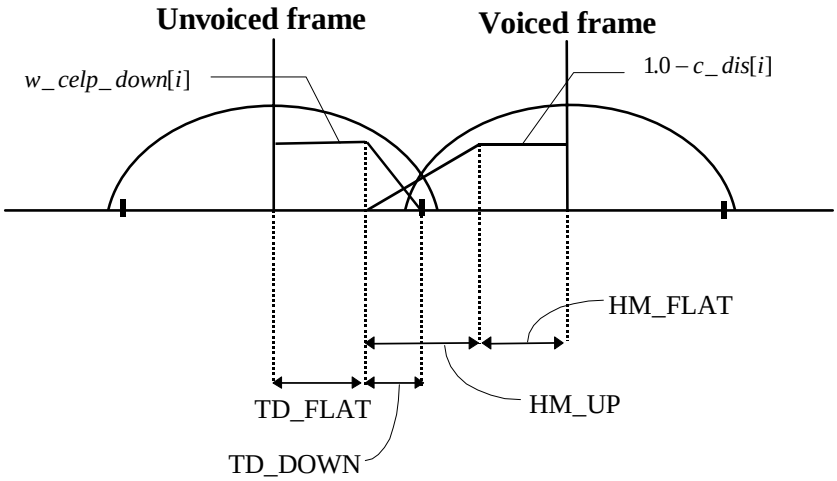


Figure 2.11 — Synthesis window for unvoiced/voiced

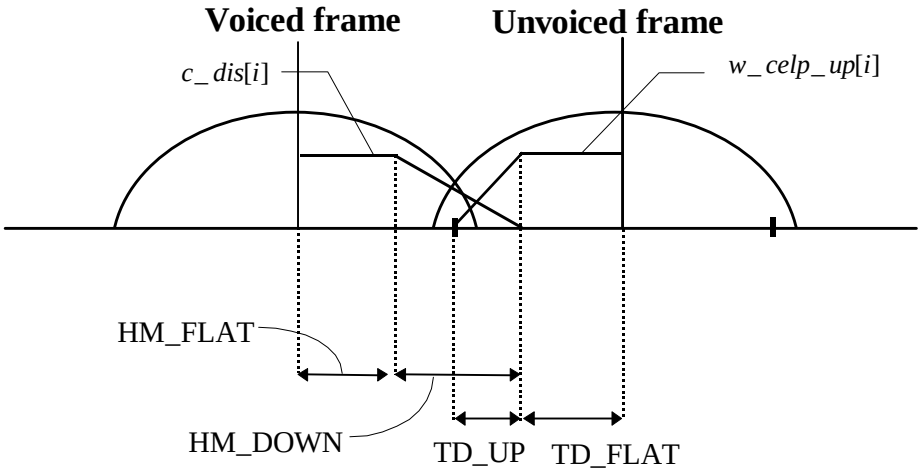


Figure 2.12 — Synthesis window for voiced/unvoiced

The first half of the unvoiced excitation signal, $suv[i]\,(0 \le i < FRM/2)$, is fed into the LPC synthesis filter

$$H_1(z) = \frac{1}{\sum_{n=0}^{P} alpha1[n]\, z^{-n}}$$

where $alpha1[]$ are linear predictive coefficients converted from the de-quantized LSPs of the previous frame, $lsp1[]$. When the second half of unvoiced excitation signal, $suv[i]\,(FRM/2 \le i < FRM)$, is fed into the LPC synthesis filter, the transfer function is switched to:

$$H_2(z) = \frac{1}{\sum_{n=0}^{P} alpha2[n]\, z^{-n}}$$

where $alpha2[]$ are linear predictive coefficients converted from the de-quantized LSPs of the current frame, $lsp2[]$.

For low delay mode, since the decode interval is shifted by LD_LEN samples, unvoiced excitation signal, $suv[i]\,(0 \le i < FRM/2 - LD\_LEN)$, is fed into the LPC synthesis filter $H_1(z)$ and unvoiced excitation signal, $suv[i]\,(FRM/2 - LD\_LEN \le i < FRM)$, is fed into the LPC synthesis filter $H_2(z)$.

Output of the LPC synthesis filter is fed into the postfilter described in subclause 2.B.1.3.2.

The output signal is in the range from -32768 to 32767.

2.5.8 Variable rate decoder

2.5.8.1 Tool description

This subclause describes tools for variable rate decoding with HVXC core. This tool allows HVXC to operate at variable bit rates, where the average bit rate is reduced to 1.2 - 1.7 kbps for typical speech material. The major part of the algorithm is composed of "background noise decoding", where only the mode bits are received during

the "background noise mode", and unvoiced frame is received with certain period of time for background noise generation.

2.5.8.2 Definitions

Definitions of constants

BGN_INTVL : maximum background noise interval (=8)

Definitions of variables

$idVUV$: V/UV decision of the current frame

$prevLSP1$: previously transmitted LSP vector

$prevLSP2$: previously transmitted LSP vector before $prevLSP1$

$bgnCnt$: frame count of successive “Background Noise” frames

2.5.8.3 Decoding process

$idVUV$ is a parameter that has the result of V/UV decision and defined as;

$$idVUV = \begin{cases} 0 & \text{Unvoiced Speech} \\ 1 & \text{Background noise interval} \\ 2 & \text{Mixed voiced speech} \\ 3 & \text{Voiced speech} \end{cases}$$

Using the background noise detection method, variable rate coding is carried out based on fixed bit rate 2kbps HVXC.

Table 2.64 — Bit allocations of the encoded parameters for the variable bit rate mode

Mode (idVUV)	Background Noise (1)	UV (0)	MV (2),V (3)
V/UV	2 bit/20 msec	2 bit/20 msec	2 bit/20 msec
LSP	0 bit/20 msec	18 bit/20 msec	18 bit/20 msec
Excitation	0 bit/20 msec	8 bit/20 msec (gain only)	20 bit/20 msec (Pitch & harmonic spectral parameters)
Total	2 bit/20 msec 0.1 kbps	28 bit/20 msec 1.4 kbps	40 bit/20 msec 2.0 kbps

For “Mixed voiced speech” and “Voiced speech” ($idVUV = 2,3$), the same decoding method as fixed bit rate mode is used.

In the decoder, two sets of LSP parameters, $prevLSP1$ and $prevLSP2$, are hold where $prevLSP1$ represents previously transmitted LSP parameters and $prevLSP2$ represents previously transmitted LSP parameters before $prevLSP1$. For “Background Noise” frame ($idVUV = 1$), VXC decoder is used in the same manner as UV frame, but no LSP parameters are sent. LSP parameters generated by linearly interpolating $prevLSP1$ and $prevLSP2$ are used for LPC synthesis, and the same gain index as the previous frame is used for excitation generation of VXC decoding. During the period of "Background Noise" frame, “Unvoiced speech (UV)” frame is inserted every ($BGN\_INTVL + 1$) (=9) frames to transmit background noise parameters. This UV frame may or may not be a real UV frame of the beginning of speech bursts. Whether or not the frame is real UV is judged by the transmitted gain index. If the gain index is smaller than or equal to that of previous one+2, then this UV frame is regarded as "Background Noise" frame, and therefore the previously transmitted LSP vector (= $prevLSP1$) is used to keep the smooth variation of LSP parameters, otherwise; currently transmitted LSPs are used as a real UV frame. The gain indices are sorted according to the magnitudes. If “Background Noise” mode is selected again, then linearly interpolated LSPs using $prevLSP1$ and $prevLSP2$ are used.

For both “Unvoiced speech” and “Background Noise” frame ($idVUV = 0,1$), Gaussian noise with a unit energy is used for excitation of VXC decoding (in place of stochastic shape codevector for VXC decoding).

Figure 2.13 shows an example. Suppose that Frame #0 and Frame #1 are “Unvoiced speech” frame, and Frame #2…Frame #9 are “Background Noise” frame. During decoding of Frame #2…Frame #9, $prevLSP1$ and $prevLSP2$ are set as: $prevLSP1 = LSP(1)$ and $prevLSP2 = LSP(0)$ and LSP vector of Frame# i , $LSP(i)$ $(2 \le i \le 9)$, are generated as

$$LSP(i) = \frac{prevLSP2 \times (2 \times BGN\_INTVL - 2 \times bgnCnt - 1) + prevLSP1 \times (2 \times bgnCnt + 1)}{2 \times BGN\_INTVL}$$

where BGN_INTVL is maximum background noise interval (=8) and $bgnCnt$ is frame count of successive “Background Noise” frames.

In this example, $bgnCnt = 0$ for Frame #2, $bgnCnt = 1$ for Frame #3, …, $bgnCnt = 7$ for Frame #9. As for gain index of VXC decoding during Frame #2…Frame #9, the gain index of Frame #1 is used. When parameters of Frame #10 are received, $prevLSP1$ and $prevLSP2$ are updated as: $prevLSP1 = LSP(10)$ and $prevLSP2 = LSP(1)$. For decoding of Frame #10, gain index is first checked whether or not it is greater than “Frame #1’s index value + 2”. If yes, Frame #10 is decoded as usual UV frame; otherwise it is decoded as “Background Noise” frame and $LSP(1)$ is used instead of $LSP(10)$ while received gain index is used for both cases.

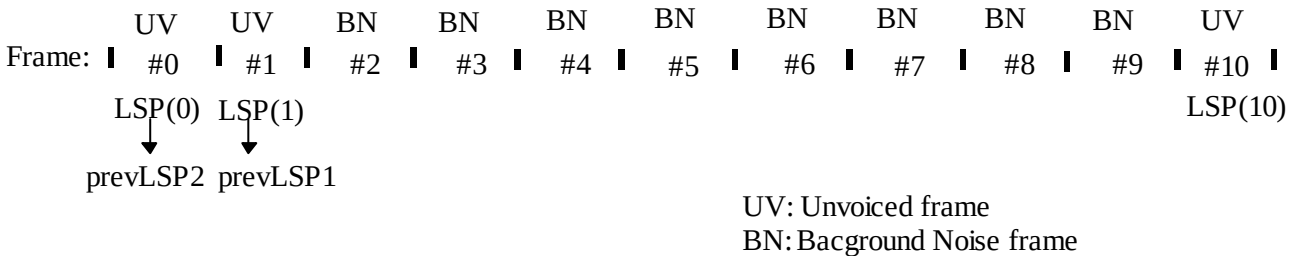


Figure 2.13 — Decoder parameter generation for background noise interval

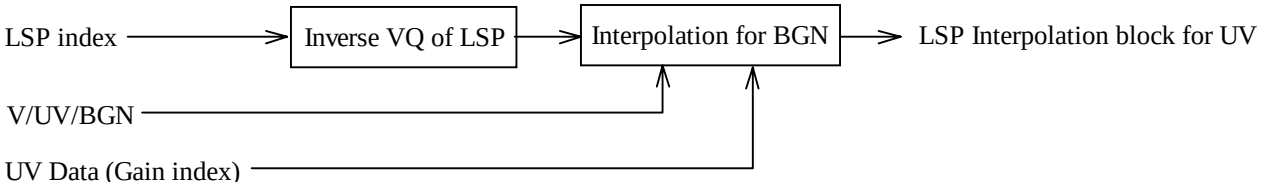


Figure 2.14 — Background noise decoder for variable rate

2.5.9 Extension of HVXC variable rate mode

2.5.9.1 Tool description

In subclause 2.5.8, variable bitrate mode based on 2 kbit/s mode is described. Here the operation of the variable bitrate mode of 4.0 kbit/s maximum is described.

In the fixed bitrate mode, we have 2 bit V/UV decision that is:

VUV = 3 : full voiced, VUV = 2 : mixed voiced, VUV=1 : mixed voiced, VUV = 0 : unvoiced.

When the operating mode is variable bitrate mode, VUV = 1 indicates “Background noise” status instead of “mixed voiced”. The current operating mode is defined by “HVXCconfig()” and decoder knows whether it’s variable or fixed rate mode and can understand the meaning of VUV = 1. In the “variable rate coding”, bit assignment is varied depending on Voiced/unvoiced decision and bitrate saving is obtained mostly by reducing the bit assignment for Unvoiced speech (VUV = 0) segment. When VUV = 0 is selected, then it is checked whether the segment is real “unvoiced speech” or “background noise” segments. If it is declared to be “background noise”, then VUV is changed to 1 and bit assignment to the frame is further reduced. During the “background noise” mode, only the mode bits or noise update frame is transmitted according to the change of the background noise characteristics. Using this variable rate mode, average bitrate is reduced to 56-85% of the fixed bitrate mode depending on the source items.

2.5.9.2 Definitions

Definitions of constants

NUM_SUBF1: the number of subframes in one frame(=2)

NUM_SHAPE_L0: the number of codebook index(=64)

BGN_INTVL: background noise update interval(=12)

Definitions of variables

prevLSP1: transmitted LSP parameters

prevLSP2: transmitted LSP parameters before prevLSP1

qLsp: LSP to be used for decoding operation of the current frame

bgnIntval: a counter which counts the number of consecutive background noise frames

rnd: a randomly generated integer value between –3 and 3

2.5.9.3 Transmission Payload

Transmission payloads with four different bitrates are used depending on V/UV decision and the result of background noise detection. VUV flag and UpdateFlag indicate the type of transmission payloads.

VUV is a parameter that has the result of V/UV decision and defined as;

$$VUV = \begin{cases} 0 & \text{Unvoiced speech} \\ 1 & \text{Background noise interval} \\ 2 & \text{Voiced speech 1} \\ 3 & \text{Voiced speech 2} \end{cases}$$

To indicate whether or not the frame marked “VUV=1” is noise update frame, a parameter “UpdateFlag” is introduced. UpdateFlag is used only when VUV=1.

$$\text{UpdateFlag} = \begin{cases} 0 & \text{not noise update frame} \\ 1 & \text{noise update frame} \end{cases}$$

If UpdateFlag is 0, the frame is not noise update frame, and if UpdateFlag is 1, the frame is noise update frame. The first frame of the “Background noise” mode is always classified as the noise update frame. In addition, if the gain or spectral envelope of the background noise frame is changed, a noise update frame is inserted.

At the noise update frame, the average of LSP parameters over the last 3 frames is computed and coded as LSP indices in the encoder. In the same manner, the average of Celp gain over the last 4 frames (8 subframes) is computed and coded as Celp gain index.

During the background noise interval (VUV = 1), LSP parameters and excitation parameters are sent only when noise update frame is selected (UpdateFlag = 1). Decoder output signals for background noise interval are generated using the LSP and excitation parameters transmitted at noise update frames.

If the current frame or the previous frame is “Background noise” mode, differential mode in LSP quantization is inhibited in the encoder, because LSP parameters are not sent during “Background noise” mode and inter frame coding is not possible.

Using the background noise detection method described above, variable rate coding is carried out based on fixed bitrate 4 kbit/s HVXC. The bitrate at each mode is shown below.

Mode(VUV)	Back Ground Noise(1)		UV(0)	V(2,3)
	UpdateFlag=0	UpdateFlag=1		
V/UV UpdateFlag LSP Excitation	2bit/20msec 1bit/20msec 0bit/20msec	2bit/20msec 1bit/20msec 18bit/20msec 4bit/20msec (gain only)	2bit/20msec 0bit/20msec 18bit/20msec 20bit/20msec	2bit/20msec 0bit/20msec 26bit/20msec 52bit/20msec
Total	3bit/20msec 0.15 kbit/s	25bit/20msec 1.25 kbit/s	40bit/20msec 2.0 kbit/s	80bit/20msec 4.0 kbit/s

2.5.9.4 Decoding Process

In the decoder, voiced frame (VUV = 2,3) is processed in the same manner as 4 kbit/s fixed bitrate mode, and unvoiced frame (VUV = 0) is processed in the same manner as 2 kbit/s fixed bitrate mode. When the background noise mode is selected (VUV=1), decoder output signal is generated in the same manner as unvoiced speech at 2 kbit/s fixed bitrate mode. The decoder parameters for back ground noise interval are generated by using the parameters transmitted at noise update frames (VUV = 1, UpdateFlag = 1) and sometimes at preceding unvoiced

frames (VUV = 0). The subclauses below show how to generate the decoder parameters for back ground noise interval.

2.5.9.4.1 LSP decoding

In the decoder, two sets of previously transmitted LSP parameters, prevLSP1 and prevLSP2, are held.

prevLSP1: transmitted LSP parameters

prevLSP2: transmitted LSP parameters before prevLSP1

"Background noise" mode occurs only after "unvoiced" or "background noise" mode. When the "background noise" mode is selected, LSP parameters are transmitted only when the frame is "noise update frame" (UpdateFlag = 1). If new LSP parameters are transmitted, prevLSP1 is copied to prevLSP2 and newly transmitted LSPs are copied to prevLSP1 regardless of VUV decision.

LSP parameters for each frame during the "background noise" mode are generated by the interpolation between prevLSP1 and prevLSP2 using the equation:

$$qLsp(i) = ratio \cdot prevLsp1(i) + (1 - ratio) \cdot prevLsp2(i) \cdots i = 1..10 \quad (2.5.9.1)$$

where

$$ratio = \frac{2 \cdot (bgnIntval + rnd) + 1}{2 \cdot BGN\_INTVL} \quad (2.5.9.2)$$

qLsp(i) is the i-th LSP to be used for decoding operation of the current frame, prevLsp1(i) is the i-th LSP of prevLSP1, prevLsp2(i) is the i-th LSP of prevLSP2 ($1 \leq i \leq 10$). In this equation, bgnIntval is a counter which counts the number of consecutive background noise frames, and is reset to 0 at the receipt of background noise update frame. BGN_INTVL(=12) is a constant, and rnd is a randomly generated integer value between –3 and 3. If counter bgnIntval reaches BGN_INTVL, bgnIntvl is set to BGN_INTVL-1, and if the ratio obtained by the equation (2.5.9.2) is smaller than 0 or greater than 1, the value of rnd is set to 0 and ratio is recomputed.

2.5.9.5 Excitation generation

During the period of "background noise" mode, the Gain index (VX_gain[0]) transmitted in the noise update frame is used for all the subframes, the values of Shape index (VX_Shape1[0,1]) are randomly generated between 0 and NUM_SHAPE_L0-1. These excitation parameters are used with the interpolated LSP parameters as described above to generate the signals of background noise mode.

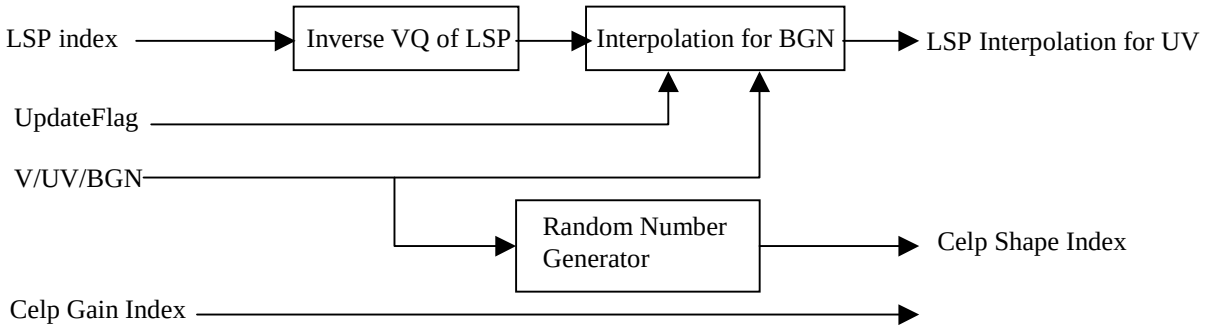


Figure 2.15 — Additional diagram for variable rate decoder

Annex 2.A
(informative)

HVXC Encoder tools

2.A.1 Overview of encoder tools

Figure 2.A.1 shows the overall structure of the encoder. Speech input at a sampling rate of 8 kHz is formed into frames with a length and interval of 256 and 160 samples, respectively. LPC analysis is carried out using windowed input data over one frame. LPC residual signals are computed by the inverse filtering of input data using quantized and interpolated LSP parameters. The residual signals are then fed into the pitch and spectral magnitude estimation block, where the spectral envelopes for LPC residual are estimated in the same manner as in the MBE coder except that only a two-bit V/UV decision is used per frame. The spectral envelope for voiced segment is then vector quantized with weighted distortion measure. For unvoiced segment, a closed loop search for the vector excitation coding is carried out. Detailed configurations are described below.

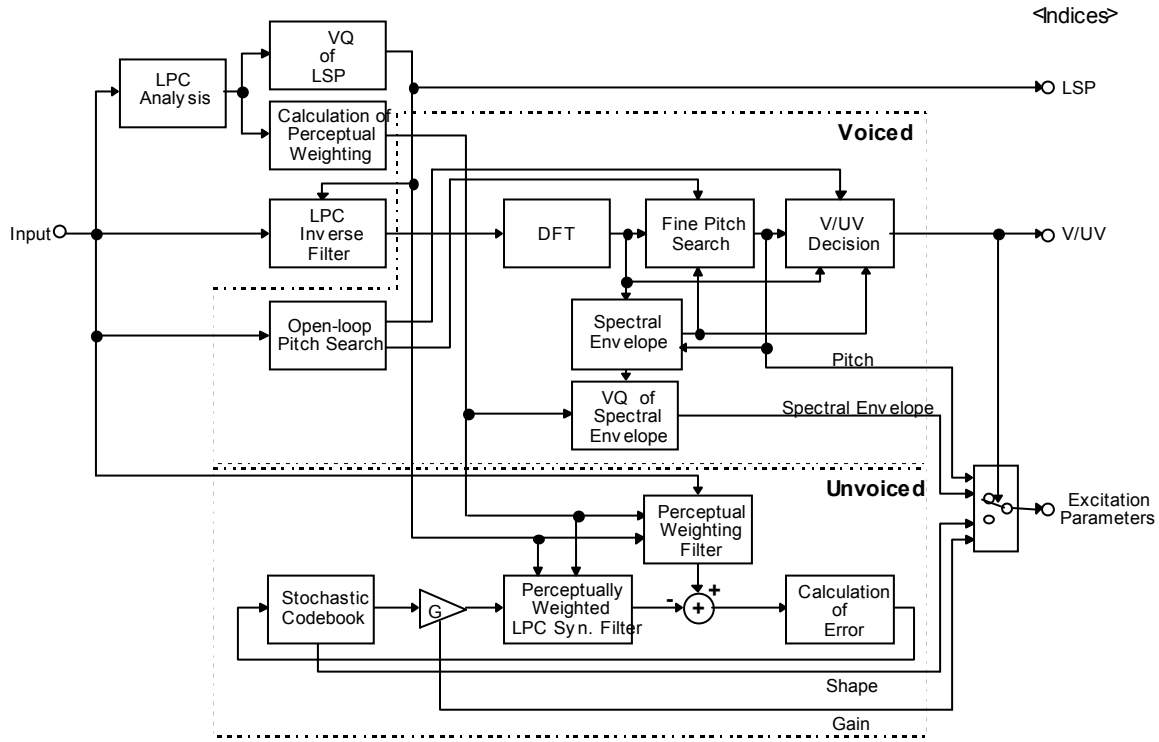


Figure 2.A.1 — Blockdiagram of the HVXC encoder

2.A.2 Normalization

2.A.2.1 Tool description

The normalization process is composed of three operations, that is, LPC analysis, LSP parameter quantization, and inverse filtering. These operations are described below.

2.A.2.2 Normalization process

2.A.2.2.1 LPC analysis

10th order LPC coefficients are computed for every frame, using Hamming-windowed input signals by autocorrelation method.

2.A.2.2.2 LSP quantization

The same LSP quantizer as that of the narrowband CELP is used.

LPC coefficients are first converted to Line Spectral Pair (LSP) parameters. LSP parameters are then quantized with a Vector Quantization (VQ). In case of the base layer, there are two methods for quantizing the LSPs as described in the decoding section; a two-stage VQ without interframe prediction, and a combination of the VQ and the inter-frame predictive VQ. At the encoding process, both methods are used to quantize the LSPs and one of them is selected by comparing the quantization error. The quantization error is calculated as a weighted euclidean distance.

In the case of the enhancement layer, a 10-dimensional vector quantizer, which has an 8 bit codebook, is added to the bottom of the current LSP quantizer scheme of the 2.0 kbps coder. The bit rate of LSPs is increased from 18 bits/20 msec to 26 bits/20 msec.

The encoding process of the base layer is as follows.

The weighting coefficients (*w[]*) are,

$$w[\mathrm{i}] = \begin{cases} \dfrac{1}{lsp[0]} + \dfrac{1}{lsp[1] - lsp[0]} & (i = 0) \\ \dfrac{1}{lsp[i] - lsp[i-1]} + \dfrac{1}{lsp[i+1] - lsp[i]} & (0 < i < Np - 1) \\ \dfrac{1}{lsp[Np-1] - lsp[Np-2]} + \dfrac{1}{1.0 - lsp[Np-1]} & (i = Np - 1) \end{cases}$$

where *Np* is the LP analysis order and *lsp[]*s are the converted LSPs.

```
w_fact = 1.;
for (i = 0; i < 4; i++) w[i] *= w_fact;
for (i = 4; i < 8; i++) {
    w_fact *= .694;
    w[i] *= w_fact;
}
for (i = 8; i < 10; i++) {
    w_fact *= .510;
    w[i] *= w_fact;
}
```

The first stage quantizer is the same for each quantization method. The LSPs are quantized by using a vector quantizer and corresponding index is stored in **LSP1** . In order to carry out delayed decision, plural indices are stored as candidates for the second stage. The quantization error in the first stage *err1*[] is given by:

$$err1[n] = \sum_{i=0}^{\dim-1} \left\{ \left(lsp[sp+i] - lsp\_tbl[n][m][i] \right)^2 \cdot w[sp+i] \right\} \qquad n = 0$$

where *n* is the split vector number, *m* is the index of the candidate split vector , sp is the starting LSP order of the n-th split vector and dim is the dimension of the n-th split vector. (*lsp_tbl[][][]* is shown in Annex 2.E)

Table 2.A.1 — Starting order and dimension of the first stage LSP vector

Split vector number: n	Starting LSP order: sp	Dimension of the vector: dim
0	0	10

In the second stage, above-mentioned two quantization methods, which are two-split vector quantizer, are applied respectively. Total quantization errors in the second stage are calculated for all combinations of the first stage candidates and the second stage candidates and the one which has the minimum error is selected. As a result, indices of the first stage are determined and corresponding indices and signs for the second stage are stored in **LSP2** and **LSP3**. The flag which indicates the selected quantization method is also stored in **LSP4**. The quantization error in the second stage *err2_total* is given by:

VQ without interframe prediction:

$$err2\_total = err2[0] + err2[1]$$

$$err2[n] = \sum_{i=0}^{\dim-1} \left\{ \left(lsp\_res[sp+i] - sign[n] \cdot d\_tbl[n][m][i] \right)^2 \cdot w[sp+i] \right\} \qquad n = 0,1$$

$$lsp\_res[sp+i] = lsp[sp+i] - lsp\_first[sp+i]$$

where *lsp_first*[] is the quantized LSP vector of the first stage, *n* is the split vector number, *m* is the index of the candidate split vector , sp is the starting LSP order of the n-th split vector and dim is the dimension of the n-th split vector. (*d_tbl[][][]* is shown in Annex 2.E.)

VQ with interframe prediction:

$$err2\_total = err2[0] + err2[1]$$

$$err2[n] = \sum_{i=0}^{\dim-1} \left\{ \left(lsp\_pres[sp+i] - sign[n] \cdot pd\_tbl[n][m][i] \right)^2 \cdot w[sp+i] \right\} \qquad n = 0,1$$

$$lsp\_pres[sp+i] = lsp[sp+i] - \left\{ \left(1 - ratio\_predict\right) \cdot lsp\_first[sp+i] + ratio\_predict \cdot lsp\_previous[sp+i] \right\}$$

where *lsp_first*[] is the quantized LSP vector of the first stage, *n* is the split vector number, *m* is the index of the candidate split vector , sp is the starting LSP order of the n-th split vector, dim is the dimension of the n-th split vector and ratio_predict = 0.7. (*pd_tbl[][][]* is shown in Annex 2.E.)

Table 2.A.2 — Starting order and dimension of the second stage LSP vector

Split vector number: n	Starting LSP order: sp	Dimension of the vector: dim
0	0	5
1	5	5

The quantized LSPs *lsp_current[]* are stabilized in order to ensure stability of the LPC synthesis filter which is derived from the quantized LSPs. The quantized LSPs are arranged in ascending order, having a minimum distance between adjacent coefficients.

```
for (i = 0; i < LPCORDER; i++) {
   if (lsp_current[i] < min_gap) lsp_current[i] = min_gap;
}
for (i = 0; i < LPCORDER-1; i++) {
   if (lsp_current[i+1]-lsp_current[i] < min_gap) {
      lsp_current[i+1] = lsp_current[i]+min_gap;
   }
}
for (i = 0; i < LPCORDER; i++) {
   if (lsp_current[i] > 1-min_gap) lsp_current[i] = 1-min_gap;
}
for (i = LPCORDER-1; i > 0; i--) {
   if (lsp_current[i]-lsp_current[i-1] < min_gap) {
      lsp_current[i-1] = lsp_current[i]-min_gap;
   }
}


for (i = 0; i < LPCORDER; i++){
   qLsp[i] = lsp_current[i];
}
```

where min_gap = 4.0/256.0

After the LSP encoding process, the current LSPs have to be stored in memory, since they are used for prediction at the next frame.

```
for (i = 0; i < LPCORDER; i++) {
     lsp_previous[i] = lsp_current[i];
}
```

It must be noted that the stored LSPs *lsp_previous[]* must be initialized as described below when the whole of the encoder is initialized.

```
for (i = 0; i < LPCORDER; i++) {
     lsp_previous[i] = (i+1) / (LPCORDER+1);
}
```

The third stage for the enhancement layer (4.0 and 3.7 kbps) has 10-dimensional VQ structure. The error between the quantized version of the base layer and the original version is quantized and corresponding index is stored in **LSP5**.

After the quantization, the LSPs of the enhancement layer qLsp[] are stabilized again.

```
for (i = 0; i < 2; i++)
{
   if (qLsp[i + 1] - qLsp[i] < 0)
   {
       tmp = qLsp[i + 1];
       qLsp[i + 1] = qLsp[i];
       qLsp[i] = tmp;
   }

   if (qLsp[i + 1] - qLsp[i] < THRSLD_L)
   {
       qLsp[i + 1] = qLsp[i] + THRSLD_L;
   }
```

```
}


for (i = 2; i < 6; i++)
{
   if (qLsp[i + 1] - qLsp[i] < THRSLD_M)
   {
       tmp = (qLsp[i + 1] + qLsp[i]) / 2.0;
       qLsp[i + 1] = tmp + THRSLD_M / 2.0;
       qLsp[i] = tmp - THRSLD_M / 2.0;
   }
}


for (i = 6; i < LPCORDER - 1; i++)
{
   if (qLsp[i + 1] - qLsp[i] < 0)
   {
       tmp = qLsp[i + 1];
       qLsp[i + 1] = qLsp[i];
       qLsp[i] = tmp;
   }

   if (qLsp[i + 1] - qLsp[i] < THRSLD_H)
   {
       qLsp[i] = qLsp[i + 1] - THRSLD_H;
   }
}
```

where, THRSLD_L = 0.020, THRSLD_M = 0.020 and THRSLD_H = 0.020

Table 2.A.3 — Configutation of the multistage LSP VQ

1st stage	10 LSP VQ	5bits
2nd stage	(5+5)LSP VQ	(7+5+1)bits
3rd stage	10LSP VQ	8bits

2.A.2.2.3 LPC inverse filter

LSPs are converted to alpha parameters to form a LPC inverse filter in a direct form. LPC residual signals are then computed by inverse filtering the input signal. Only the current frame's quantized LPC is used without any interpolation for the inverse filtering to compute the LPC residual signal. The residual signal is then 256 point Hamming windowed to compute the power spectrum. The transfer function of the LPC inverse filter is,

$$A(z) = \sum_{n=0}^{P} \alpha_n z^{-n}$$

where $P = 10$ and α_0 = 1. LPC coefficients α_n stay unchanged during the computation of residual samples of one frame length (256 samples).

2.A.3 Pitch estimation

2.A.3.1 Tool description

To obtain the first estimation of the pitch lag value, the autocorrelation values of the LPC residual signals are computed. Based on the lag values which give the peaks in autocorrelation, the open loop pitch is estimated. Pitch tracking is carried out in the process of pitch estimation so that the estimated pitch gets more reliable.

2.A.3.2 Pitch estimation process

In the low delay mode encoder, pitch tracking is conducted using only current and past frames to keep encoder delay 26 ms. When the normal delay mode is used, pitch tracking uses one frame ahead and the encoder delay becomes 46 ms.

2.A.3.3 Pitch tracking

For the low delay mode of HVXC, pitch tracking algorithm does not use the pitch value of the future (look ahead) frame. The pitch tracking algorithm operates based on reliable pitch "rblPch", V/UV decision of the previous frame "prevVUV", and past/current pitch parameters.

The basic operation is as follows:

- When prevVUV != 0 and rblPch != 0, pitch is tracked based on rblPch. However, the pitch value of the previous frame has higher priority.
- When prevVUV = 0 and rblPch != 0, pitch is tracked based on rblPch.
- When prevVUV != 0 and rblPch = 0, tracking is conducted based on the pitch of the previous frame.
- When prevVUV = 0 and rblPch = 0, current pitch parameter is simply used.

prevVUV !=0 represents Voiced, and prevVUV = 0 represents Unvoiced status respectively.

By this strategy, pitch tracking is carried out without any look ahead. Source code of the pitch tracking is shown below.

```
typedef struct
{
    float  pitch;     /* pitch */
    float  prob;      /* 1st peak devided by 2nd peak of autocorrelation */
    float  r0r;       /* 1st peak of autocorrelation  - modified */
    float  rawR0r;    /* 1st peak of autocorrelation - raw */
    float  rawPitch;  /* pitch with no tracking */
}
NgbPrm;

static int NearPitch(
float   p0,
float   p1,
float   ratio)
{
      return((p0 * (1.0 - ratio) < p1) && (p1 < p0 * (1.0 + ratio)));
}

static float TrackingPitch(
NgbPrm  *crntPrm,  /* current parameter */
NgbPrm  *prevPrm,  /* previous  parameter */
int     *scanlimit,  /* Number of autocorrelation peaks */
float   *ac,    /* Autocorrelation */
int  peakPos[PEAKMAX],  /* Position of autocorrelation peaks */
int  prevVUV)     /* V/UV of ptrevious frame */
```

```
{
    float  kimete;
    float  pitch;
    int    i;
    static float   prevRawp= 0.0;
    int    st0, st1, st2;
    float  stdPch;
    static float   rblPch = 0.0;
    static float   prevRblPch = 0.0;

    rblPch = global_pitch;
    if (prevVUV != 0 && rblPch != 0.0)
    {
   st0 = Ambiguous(prevPrm->pitch, rblPch, 0.11);
   st1 = Ambiguous(crntPrm->pitch, rblPch, 0.11);
     if(!(st0 || st1))
   {
       if (NearPitch(crntPrm->pitch, prevPrm->pitch, 0.2))
     pitch = crntPrm->pitch;
       else if (NearPitch(crntPrm->pitch, rblPch, 0.2))
     pitch = crntPrm->pitch;
       else if (NearPitch(prevPrm->pitch, rblPch, 0.2))
       {
     if (crntPrm->r0r > prevPrm->r0r && crntPrm->prob > prevPrm->prob)
         pitch = crntPrm->pitch;
     else
         pitch = prevPrm->pitch;
       }
       else
     pitch = GetStdPch2Elms(crntPrm, prevPrm, scanlimit, peakPos, ac);
   }
   else if (!st0)
   {
       if (NearPitch(prevPrm->pitch, crntPrm->pitch, 0.2))
     pitch = crntPrm->pitch;
       else if ((gpMax * 1.2 > crntPrm->pitch) &&
                           NearPitch(prevPrm->rawPitch, crntPrm->pitch, 0.2))
     pitch = crntPrm->pitch;
       else
     pitch = prevPrm->pitch;
   }
   else if (!st1)
   {
       if ((crntPrm->rawPitch != crntPrm->pitch) &&
     NearPitch(crntPrm->rawPitch, prevPrm->rawPitch, 0.2))
     pitch = crntPrm->rawPitch;
       else
     pitch = crntPrm->pitch;
   }
   else
   {
       if (NearPitch(prevPrm->pitch, crntPrm->pitch, 0.2))
     pitch = crntPrm->pitch;
       else
     pitch = rblPch;
   }
    }
    else if (prevVUV == 0 && rblPch != 0.0)
    {
   st1 = Ambiguous(crntPrm->pitch, rblPch, 0.11);
   if (!st1)
       pitch = crntPrm->pitch;
   else
       pitch = rblPch;
    }
    else if (prevVUV != 0 && rblPch == 0.0)
    {
   st1 = Ambiguous(crntPrm->pitch, prevPrm->pitch, 0.11);
```

```
   if (!st1)
       pitch = GetStdPch2Elms(crntPrm, prevPrm, scanlimit, peakPos, ac);
   else
   {
       if (prevPrm->r0r < crntPrm->r0r)
      pitch = crntPrm->pitch;
       else
      pitch = prevPrm->pitch;
   }
    }
    else
   pitch = crntPrm->pitch;

    crntPrm->pitch = pitch;
    prevRblPch = rblPch;
    prevRawp = pitch;
    return(pitch);
}
```

2.A.4 Harmonic magnitudes extraction

2.A.4.1 Tool description

Harmonic magnitudes extraction consists of two steps: fine pitch search and estimation of the spectral envelope. The operation of each step is described below.

2.A.4.2 Harmonic magnitudes extraction process

2.A.4.2.1 Fine pitch search

Using the open loop integer pitch lag, the fractional pitch lag value is estimated here. The step size of the fraction is 0.25. This is carried out by minimizing the error between the synthesized spectrum and original spectrum. Here, pitch lag value and spectral harmonic magnitudes are estimated simultaneously. Estimated pitch lag value, pch, is transmitted to the decoder as **Pitch**.

$$Pitch = (int)(pch - 20.0)$$

2.A.4.2.2 Estimation of spectral envelope

A 256 point DFT is applied to the LPC residual signals to obtain the original spectrum. Using the original spectrum, estimation of the spectral envelope is carried out in the part of the above mentioned fine pitch search. The spectral envelope is a set of spectral magnitudes estimated at each harmonic. This magnitude estimation is carried out by computing an optimal amplitude A_m using pre-defined basis function $E(j)$, and original spectrum $X(j)$. Let a_m and b_m be the indices of DFT coefficient of lower and higher boundary of the m-th band respectively, covering m-th harmonic band. The amplitude estimation error ε_m is defined as:

$$\varepsilon_m = \sum_{j=a_m}^{b_m} \left(\left| X(j) \right| - \left| A_m \right| \left| E(j) \right| \right)^2$$

Solving

$$\frac{\partial \varepsilon_m}{\partial |A_m|} = 0$$

gives

$$|A_m| = \frac{\sum_{j=a_m}^{b_m} |X(j)||E(j)|}{\sum_{j=a_m}^{b_m} |E(j)|^2}$$

This value is defined as spectral magnitude. Basis function $E(j)$ can be obtained by DFT of 256 point Hamming window.

2.A.5 Perceptual weighting

2.A.5.1 Tool description

The frequency response of a perceptual weighting filter is computed which is for the use of weighted vector quantization of the harmonic spectral envelope. Here, a perceptual weighting filter is derived from linear predictive coefficients α_n. The transfer function of the perceptual weighting filter is;

$$w(z) = \frac{\sum_{n=0}^{P} \alpha_n A^n z^{-n}}{\sum_{n=0}^{P} \alpha_n B^n z^{-n}}$$

where A = 0.9 B = 0.4 could be used. In this tool, the frequency response of the LPC synthesis filter $h(z)$ is also computed so that it could be incorporated where,

$$h(z) = \frac{1}{\sum_{n=0}^{P} \alpha_n z^{-n}}$$

In this tool, frequency response of $w(z)h(z)$ is computed and outputed as an array *per_weight, which could be used as diagonal components of the weighting matrices WH.

2.A.6 Harmonic VQ encoder

2.A.6.1 Tool description

The harmonic VQ encoding process consists of two steps: dimension conversion and vector quantization of the spectral envelope vectors. The operations of each step are given below

2.A.6.2 Encoding process

2.A.6.2.1 Dimension converter

The number of points which composes the spectral envelope varies depending on pitch values, since the spectral envelope is a set of the estimates of the magnitudes at each harmonic. The number of harmonics ranges from about 9 to 70.

In order to vector quantize the spectral envelope, the coder has to convert them to a constant number for a fixed-dimension VQ. A band-limited interpolation is used for the sampling frequency conversion to obtain the fixed-dimension spectral vectors. The number of points, which represent the shape of the spectral envelope, should be modified without changing the shape. For this purpose, a dimension converter for a spectral envelope by a combination of low pass filter and 1st order linear interpolator is used. An FIR low pass filter with 7 sets of coefficients, each set consisting of 8 coefficients, is used for the first stage 8-times over-sampling. The 7 sets of the filter coefficients are obtained by grouping 8 every coefficients from a windowed sinc, $coef[i]$, with the offsets of 1 through 7, where

$$coef[i] = \frac{\sin \pi(i-32)/8}{\pi(i-32)/8}(0.5 - 0.5\cos 2\pi i/64) \qquad 0 \le i \le 64$$

This FIR filtering allows decimated computation, in which only the points used at the next stage are computed. They are the left and right adjacent points of the final output of the dimension converter.

At the second over-sampling stage, 1st order linear interpolation is applied to obtain the necessary output points. In this way, we get fixed-dimension (= 44) spectral vectors.

2.A.6.2.2 Vector quantization

A fixed-dimension (= 44) spectral vector is then quantized. In the base layer, a two-stage vector quantization scheme is employed for the spectral shape together with a scalar quantizer for the gain. The weighted distortion measure D is used for the codebook search of both shape and gain.

$$D = \left\| \mathbf{WH}(\mathbf{x} - g(\mathbf{s}_1 + \mathbf{s}_2)) \right\|^2$$

where $\boldsymbol{x}$ is a source vector, $\mathbf{s}_1$ is the output of Spectral Envelope (SE) shape1 codebook, $\mathbf{s}_2$ is the output of SE shape2 codebook, and g is the output of the SE gain codebook. Corresponding indices are denoted as **SE_shape1**, **SE_shape2** and **SE_gain** respectively. The dimension of the shape codebooks is fixed (=44). The diagonal components of the matrices $\mathbf{H}$ and $\mathbf{W}$ are the magnitudes of the frequency response of the LPC synthesis filter and the perceptual weighting filter, respectively.For enhancement layer, additional vector quantizers are added to the bottom of the quantizer of the base layer. The 2.0kbps mode uses only the quantizers of the base layer.

For 4.0 kbps mode, the quantized harmonic magnitudes with fixed dimension (=44) at the base layer is first converted to the dimension of the original harmonic vector, which varies depending on the pitch value. The difference between the quantized/dimension recovered harmonic vector and the original harmonic vector is computed. This difference is then quantized with a split VQ scheme composed of four vector quantizers at the enhancement layer. The corresponding indices are **SE_shape3**, **SE_shape4**, **SE_shape5** and **SE_shape6**.

Index **SE_shape6** is not used when 3.7 kbps mode is selected.

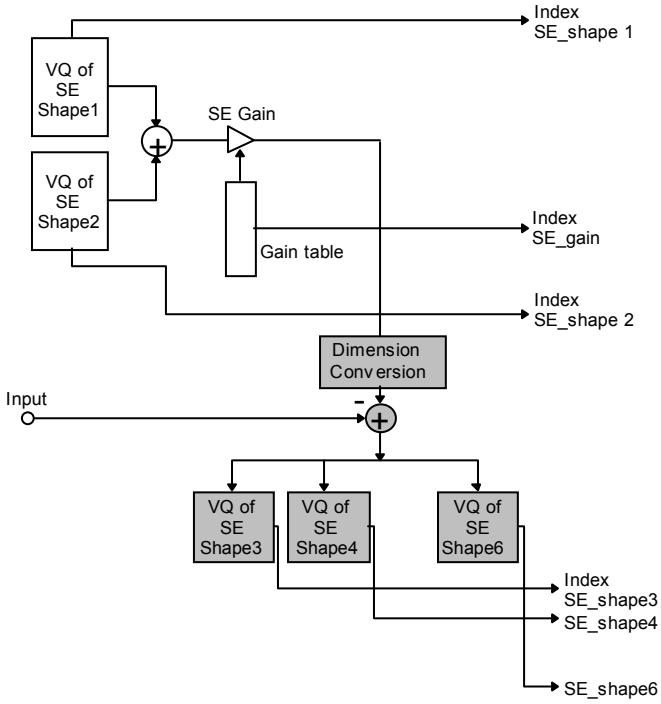


Figure 2.A.2 — Vector quantization of spectral envelope

2.A.7 V/UV decision

2.A.7.1 Tool description

A V/UV decision is made every 20 ms frame. The decision is made based on: similarity of the shape of the synthesized spectrum and original spectrum, signal power, maximum autocorrelation of LPC residual signals normalized by residual signal power, and number of zero crossing.

2.A.7.2 Encoding process

The V/UV decision is composed of three different modes, that is, Unvoiced, Mixed Voiced, and Fully Voiced.

In order to send out the information of the three modes, two bits are used for V/UV. First, a decision is made whether or not the current frame is Unvoiced. When this decision is Voiced, then the voicing strength is evaluated from the value of the normalized maximum peak of the autocorrelation of LPC residual signal. Let us denote the value r0r.

Shown below is the decision rule :

When the first decision is Unvoiced V/UV is 0 - Unvoiced

When the first decision is Voiced V/UV is 1 for r0r < TH2 - Mixed Voiced 1

V/UV is 2 for TH2<= r0r < TH1 - Mixed Voiced 2

V/UV is 3 for TH1<= r0r - Fully Voiced

Example:

TH1 = 0.7

TH2 = 0.5

The number 0,1,2,3 is send out to the decoder as index **VUV** using the two bits.

2.A.8 Time domain encoder

2.A.8.1 Tool description

When a speech segment is unvoiced, Vector Excitation Coding (VXC) algorithm is used. Figure 2.A.3 shows the overall structures of the VXC encoder. The operation of the VXC is described below.

2.A.8.2 Encoding process

LPC analysis is carried out first, and LPC coefficients (α) are then converted to LSP parameters in the same manner as in the voiced case. LSP parameters are quantized, and quantized LSPs are converted to LPC coefficients ($\hat{\alpha}$).

The perceptually weighted LPC synthesis filter $H(z)$ is expressed as:

$$H(z) = A(z)W(z)$$

where $A(z)$ is a transfer function of the LPC synthesis filter, and $W(z)$ is a perceptual weighting filter derived from LPC coefficients.

Let $x_w(n)$ be perceptually weighted input signal. Subtracting zero-input response $z(n)$ from $x_w(n)$, we obtain reference signal, $r(n)$, for the analysis-by-synthesis procedure of the VXC. Optimal shape and gain vectors are searched using the distortion measure E:

$$E = \sum_{n=0}^{N-1} \left(r(n) - g \times syn(n) \right)^2$$

where $syn(n)$ is zero-state response of $H(z)$, driven by a excitation input by only a shape vector $s(n)$, which is an output of VX_shape codebook. g is a gain, which is an output of VX_gain coodbook. N is vector dimension of VX_shape codebook.

The codebook search process for the VXC consists of two steps, that are:

1. Search $s(n)$ that maximize

$$E_s = \frac{\sum_{n=0}^{N-1} r(n) \times syn(n)}{\sqrt{\sum_{m=0}^{N-1} syn(m)^2}}$$

2. Search g that minimize

$$E_g = \left(g_{ref} - g\right)^2$$

where

$$g_{ref} = \frac{\sum_{n=0}^{N-1} r(n) \times syn(n)}{\sum_{m=0}^{N-1} syn(m)^2}$$

The quantization error $e(n)$ is computed as:

$$e(n) = r(n) - g \times syn(n)$$

When the bit-rate is 4 kbps, one more stage is used for the quantization of unvoiced segments, and $e(n)$ is used as reference input to the second stage VQ.

The operation of the second stage VQ is the same as that of the first stage VQ.

The 2.0 kbps coder uses 6 bits shape and 4 bit gain codebooks for the unvoiced excitation for every 10 msec. The corresponding indices are **VX_shape[i], VX_gain[i]** (i=0,1). The 4.0 kbps scheme adds 5 bits shape and 3 bits gain codebooks for every 5 msec at the bottom of the current quantizer. The corresponding indices are **VX_shape2[i], VX_gain2[i]** (i = 0,1,2,3). 3.7 kbps mode may use the same encoding procedure as that of 4.0kbps mode though VX_shape2[3] and VX_gain2[3] are not used in the decoder at 3.7kbps mode.

Table 2.A.4 — Configuration of the VXC codebooks

1st stage	(80dimension 6bits shape + 4bits gain) x 2
2nd stage	(40dimension 5bits shape + 3bits gain) x 4

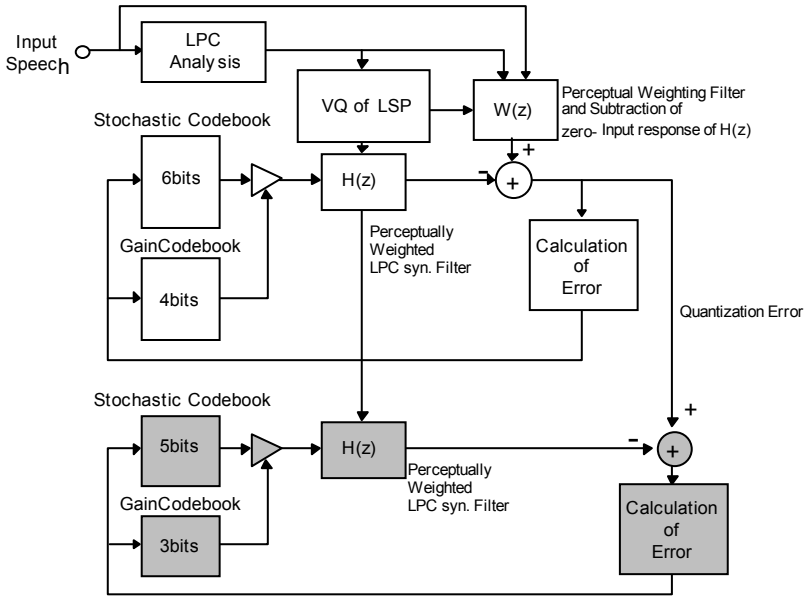


Figure 2.A.3 — Vector Excitation Coding (VXC) for unvoiced signals

2.A.9 Variable rate encoder

This subclause describes a tool for variable rate coding with the HVXC core. This tool allows HVXC to operate at variable bit rates. The major part of the algorithm is composed of "background noise interval detection", where only the mode bits are transmitted during the "background noise mode", and unvoiced frame is inserted with certain period of time to send parameters for background noise generation

In the encoding algorithm, minimum level tracker has temporal minimum level in order to adjust a threshold value by which a decision is made about whether or not the input segment is speech.

Minimum level tracking

Parameters are defined as follows:

lev: r.m.s. of a speech frame

vCont: number of continuous voiced frames

cdLev: candidate value of minimum level

prevLev: r.m.s. of a previous frame

gmlSetState: number of frames in which the candidate value is set

gmlResetState:number of frames in which the candidate value is not set after setting of minimum level

gml:minimum level

The minimum level is tracked according to the algorithm shown below.

```
if (vCont > 4)
{
    cdLev = 0.0;
```

```
    gmlSetState = 0;
    gmlResetState++;
}
else if (lev < MIN_GML)
{
    *gml = MIN_GML;
    gmlResetState = 0;
    gmlSetState = 0;
}
else if (*gml > lev)
{
    *gml = lev;
    gmlResetState = 0;
    gmlSetState = 0;
}
else if ((lev < 500.0 && cdLev * 0.70 < lev && lev < cdLev * 1.30) || lev < 100.0)
{
    if (gmlSetState > 6)
    {
        *gml = lev;
        gmlResetState = 0;
        gmlSetState = 0;
    }
    else
    {
        cdLev = lev;
        gmlResetState = 0;
        gmlSetState++;
    }
}
else if ((lev < 500.0 && prevLev * 0.70 < lev && lev < prevLev * 1.30) || lev < 100.0)
{
    cdLev = lev;
    gmlResetState = 0;
    gmlSetState++;
}
else if (gmlResetState > 40)
{
    *gml = MIN_GML;
    gmlResetState = 0;
    gmlSetState = 0;
}
else
{
    cdLev = 0.0;
    gmlSetState = 0;
    gmlResetState++;
}
```

As shown above, the minimum level is held during appropriate period of time and updated. The minimum level is always set higher than a predetermined value MIN_GML.

Background noise detection based on the minimum level

The reference level refLev is computed as,

$$refLev = A \times \max(lev, refLev) + (1.0 - A) \times \min(lev, refLev) \qquad (1)$$

Usually A is set to 0.75. Background noise detection is carried out using the refLevel derived from equation (1).

For the frames where "voiced" decision is made:

```
if (refLev < B *gml && contV < 2) {
   idVUV = 1;
}
```

For the frames where "unvoiced" decision is made:

```
if (refLev < B *gml) {    /* condition-1 */
   if (bgnCnt < 3) {
     bgnCnt++;
   }
   else {
      if (bgnIntvl < 8) {
         idVUV=1:
         bgnIntvl++;
      }
      else {
         bgnIntvl=0;
      }
   }
}
else {
   bgnCnt=0;
}
```

where B is a constant. In this case we set B=2.0

Each parameter is defined below.

countV : number of consecutive voiced frames

bgnCnt: number of frames which satisfies the condition-1

bgnIntvl: number of frames where "Background noise" mode is declared

idVUV is a parameter that has the result of V/UV decision and defined as;

$$\text{idVUV} = \begin{cases} 0 & \text{Unvoiced speech} \\ 1 & \text{Background noise interval} \\ 2 & \text{Mixed voiced speech} \\ 3 & \text{Voiced speech} \end{cases}$$

If the current frame is declared to be "Voiced", it is checked whether or not the previous frame is "Voiced". If the previous frame is "Voiced", "backgroundnoise" mode is not selected; otherwise, "background noise" mode is selected.

If the current frame is declared to be "Unvoiced", "background noise" mode is selected only after the condition-1 is satisfied for four consecutive frames. When "background noise" mode is selected for consecutive N frames, then the last frame is replaced with "Unvoiced" mode in order to transmit speech parameters which represents characteristics of the time varying back ground noise.

At this moment, N is set to 9.

Variable rate encoding:

Using the background noise detection method described above, variable rate coding is carried out based on fixed bit rate 2 kbps HVXC.

Table 2.A.5 — Bit allocations of the encoded parameters for the variable bit rate mode

Mode (idVUV)	Back Ground Noise (1)	UV (0)	MV (2), V (3)
V/UV LSP Excitation	2bit/20msec 0bit/20msec 0bit/20msec	2bit/20msec 18bit/20msec 8bit/20msec (gain only)	2bit/20msec 18bit/20msec 20bit/20msec
Total	2bit/20msec 0.1kbps	28bit/20msec 1.4kbps	40bit/20msec 2.0kbps

If the current frame or the previous frame is "Background noise" mode, differential mode in LSP quantization is inhibited in the encoder because LSP parameters are not sent during "Background noise" mode and inter frame coding is not possible.

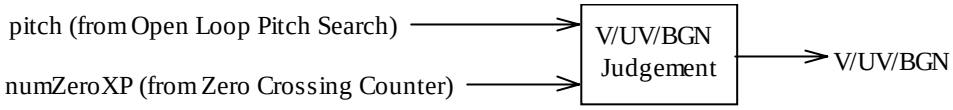


Figure 2.A.4 — Block diagram for variable rate encoding

2.A.10 Extension of HVXC variable rate encoder

In subclause 2.A.9, the encoder operation of the variable rate mode of 2.0 kbit/s maximum is described. Here in this subclause one example of the implementation of HVXC variable rate mode encoder of 4.0 kbit/s maximum is described. Basically any kind of background noise/unvoiced speech decision algorithm could be used. The change of the signal level and the spectral envelope are used to detect background noise interval from unvoiced frames. During the noise interval, it is assumed that the signal level and the shape of the spectral envelope are stable. The log squared magnitude response at low frequency range of the spectral envelope is computed from LPC cepstrum coefficients. The log squared magnitude response of the current frame is compared with the average of the log magnitude response over several past frames. If the difference is small, it is assumed that the signal is stable. When these parameters show the signal condition is stable enough, the frame is classified as “background noise interval”. If the signal characteristic is changed in certain range, it is assumed that the background noise characteristic is changed, and noise update frame is transmitted. If the signal characteristic is changed more than pre-defined range, then signal is assumed to be unvoiced speech.

2.A.10.1 Definitions

stFlag: signal stability flag
bgnCnt: background noise counter

BgnIntvl: backgorund noise interval counter
UpdateFlag : background noise update flag

2.A.10.2 RMS computation

RMS of the input signal $s[]$ is first computed to obtain minimum signal level ($\min\_rms$). If the RMS value is smaller than the predefined value that is the smallest possible speech level, and the deviation of RMS values of several frames are in certain range, then the minimum level is smoothly updated using the detected RMS value and the current $\min\_rms$. The current RMS value "rms" is then divided by current $\min\_rms$ to obtain the value "$ratio$".

$$ratio = \frac{rms}{\min\_rms}$$

This value is used for signal stability detection as described in 2.A.10.4.

2.A.10.3 Spectral comparison

The spectral envelope is computed using Linear Prediction (LP) coefficients. LP coefficients are converted to cepstrum parameters $C_L[]$. From cepstrum parameters, log squared magnitude response $\ln\left|H_L(e^{j\Omega})\right|^2$ is computed as:

$$\ln\left|H_L(e^{j\Omega})\right|^2 = 2\cdot\sum_{m=0}^{M} C_L(m)\cos(\Omega m)$$

The average level of each of the frequency bands at the n-th frame is computed as:

$$\log Amp(n,i) = \frac{1}{w}\int_{\Omega_i}^{\Omega_{i+1}} \ln\left|H_L(e^{j\Omega})\right|^2 d\Omega = \frac{1}{w}\left[2\sum_{m=1}^{M}\frac{1}{m}C_L(m)\sin(\Omega m)\right]_{\Omega_i}^{\Omega_{i+1}} \qquad \text{i = 0…3}$$

Here, ω is the band width (=500Hz), and i is the band number where 4 bands between 0 and 2kHz are computed. From the obtained log squared magnitude values of the last 4 frames, averaged values for each band is computed as:

$$aveAmp(n,i) = \frac{1}{4}\sum_{j=1}^{4}\log Amp(n-j,i)$$

Using these equations, The difference "$wdif$" between the current level and averaged level for each of the frequency band i is computed as shown below.

$$wdif = \sqrt{\frac{1}{4}\sum_{i=0}^{3}(\log Amp(n,i) - aveAmp(n,i))^2}$$

2.A.10.4 Signal stability detection

From the combination of $ratio$ and $wdif$, signal stability flag "stFlag" which has the following 3 states is computed. Naturally, the smaller these variables are, the more stable the signal state is.

$$\text{stFlag} = \begin{cases} 0 & \text{Almost stable} \\ 1 & \text{Unstable} \\ 2 & \text{A little stable} \end{cases}$$

2.A.10.5 Background noise detection

According to the variable stFlag and the V/UV decision, background noise is detected. The flowchart is shown in Figure 2.A.62.A.6. When the V/UV decision is voiced (VUV = 2,3), the frame is classified as voiced regardless as the value of stFlag. When background noise is detected in unvoiced interval (VUV = 0), VUV is set to 1. When background noise parameters have to be updated, “UpdateFlag” is set to 1. In order to secure stable operation of the background noise detection algorithm, the variables, background noise counter (bgnCnt) and background noise interval counter (bgnIntvl), are introduced. BGN_CNT and BGN_INTVL are predefined constants.

2.A.10.6 Transmitted parameters decision

VUV is a parameter that has the result of V/UV decision and defined as;

$$\text{VUV} = \begin{cases} 0 & \text{Unvoiced speech} \\ 1 & \text{Background noise interval} \\ 2 & \text{Voiced speech 1} \\ 3 & \text{Voiced speech 2} \end{cases}$$

In order to indicate whether or not the frame marked “VUV = 1” is noise update frame, a parameter “UpdateFlag” is introduced. UpdateFlag below is used only when VUV = 1.

$$\text{UpdateFlag} = \begin{cases} 0 & \text{not noise update frame} \\ 1 & \text{noise update frame} \end{cases}$$

At noise update frame, averaged LSP and Celp Gain parameters are computed.

As for LSP, raw LSPs are averaged around the last 3 frames.

$$aveLsp[n][i] = \frac{1}{3}\sum_{j=0}^{2} Lsp[n-j][i] \ , \quad \text{i = 1,...,NP}$$

where $aveLsp[n][]$ denotes the averaged LSP at the n-th frame and $Lsp[n][]$ denotes the raw LSP at the n-th frame. NP is the order of LSP. $aveLsp[n][]$ is quantized and coded, where LSP coding by differential mode is inhibited.

As for Celp gain, raw Celp gains around the last 4 frames(8 subframes) are averaged.

$$aveGain[n] = \frac{1}{8}\sum_{j=0}^{3}\sum_{k=0}^{1} Gain[n-j][k]$$

where aveGain[n] is the averaged Celp gain at the n-th frame and Gain[n][k] is the raw Celp gain at the n-th frame and the k-th subframe. The averaged Celp gain is quantized and coded in the same manner as usual Celp gain.

According to VUV, the following parameters are transmitted.

Table 2.A.6 — coded parameters for 4 kbit/s variable rate mode

Mode(VUV)	Back Ground Noise(1)		UV(0)	V(2,3)
	UpdateFlag=0	UpdateFlag=1		
V/UV UpdateFlag LSP Excitation	2bit/20msec 1bit/20msec 0bit/20msec	2bit/20msec 1bit/20msec 18bit/20msec 4bit/20msec (gain only)	2bit/20msec 0bit/20msec 18bit/20msec 20bit/20msec	2bit/20msec 0bit/20msec 26bit/20msec 52bit/20msec
Total	3bit/20msec 0.15 kbit/s	25bit/20msec 1.25 kbit/s	40bit/20msec 2.0 kbit/s	80bit/20msec 4.0 kbit/s

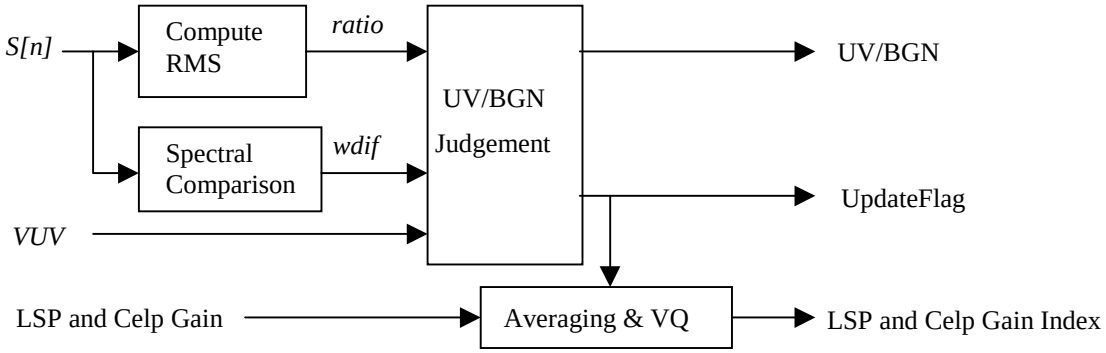


Figure 2.A.5 — Additional Blockdiagram for Encoder

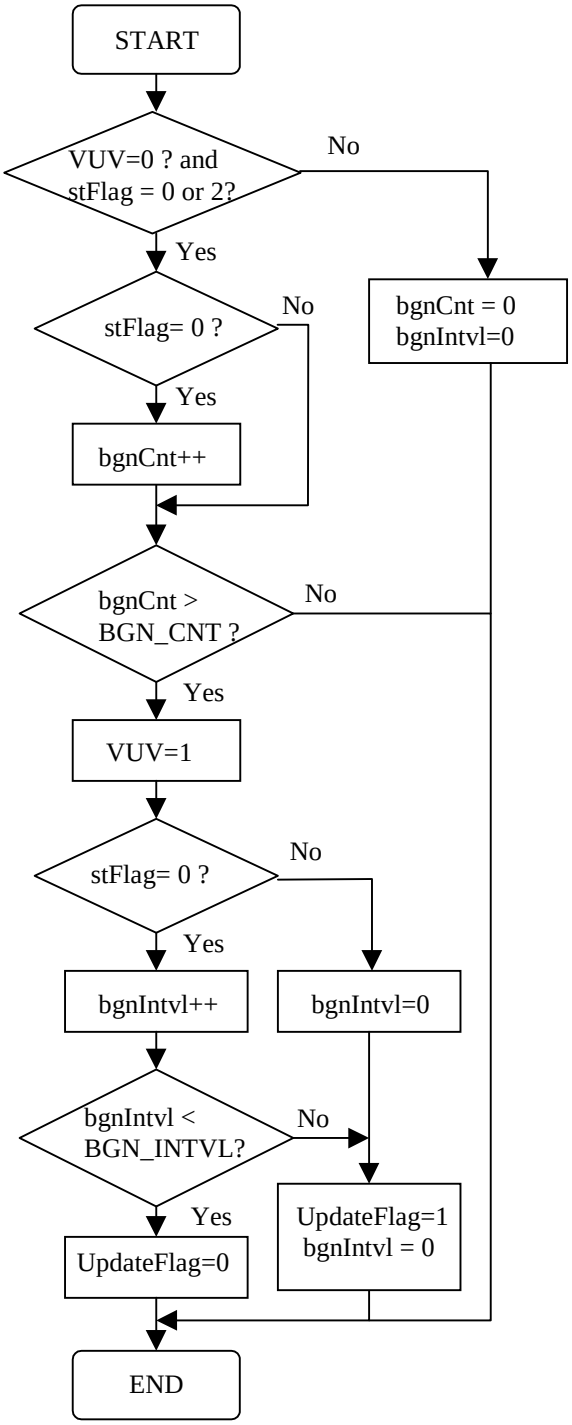


Figure 2.A.6 — Flowchart of noise detection

Annex 2.B
(informative)

HVXC Decoder tools

2.B.1 Postfilter

2.B.1.1 Tool description

The basic operation of the post-filter is to enhance the spectral formants and suppress the spectral valleys. One can use a single postfilter after voiced and unvoiced synthesized speech are added. Alternatively, one can use independent postfilters for voiced and unvoiced speech respectively. Use of independent postfilters for voiced and unvoiced signals is recommended.

In the description below, each of the voiced speech and unvoiced speech obtained is fed into the independent postfilters.

The use of a postfilter is required, however, the configuration and constants of the postfilters described here is one example and not normative, and they could be modified.

2.B.1.2 Definitions

$Pf_v(z)$: Transfer function of spectral shaping filter for voiced speech.

$Pf_{uv}(z)$: Transfer function of spectral shaping filter for unvoiced speech

r_{adj}: gain adjustment factor for spectral shaping.

$S(n)$: LPC sythesis filter $H(z)$ output.

$S_{pf}(n)$: Spectral shaping filter $Pf_v(z)$ output.

$S_{pf}'(n)$: Spectral shaped and gain adjusted output.

$S_{pf}'\_prev(n)$: $S_{pf}'(n)$ computed using previous frame's α_n and r_{adj}.

$vspeech(n)$: Postfiltered voiced speech.

$uvspeech(n)$: Postfiltered unvoiced speech.

2.B.1.3 Processing

Each of the operation of the postfilter consists of three steps, that is, spectral shaping, gain adjustment and smoothing process.

2.B.1.3.1 Voiced speech

Spectral shaping:

$$Pf_v(z) = \frac{\sum_{n=0}^{P} \alpha_n \gamma^n z^{-n}}{\sum_{n=0}^{P} \alpha_n \beta^n z^{-n}} \left(1 - \delta z^{-1}\right)$$

Output of the LPC synthesis filter s(n) is first fed into the spectral shaping filter $Pf_v(z)$, where α_n are linear predictive coefficients converted from de-quantized and linearly interpolated LSPs, which are updated every 2.5 ms. $p = 10, \gamma = 0.5, \beta = 0.8, \delta = -0.15 \times \alpha_1$ where the value of δ is limited to the range of $0 \le \delta \le 0.5$. When low delay decode mode is selected, the decode frame interval is shifted by 2.5 ms and interpolation of LSPs is carried out for the first 17.5ms of decode frame interval shown in Figure 2.9, and the latest LSPs are used for the last 2.5 ms without interpolation.

Gain adjustment:

The output of the spectral shaping filter, $S_{pf}(n)$, is then gain adjusted so that the frame gain of the input and output of the spectral shaping is unchanged. Gain adjustment is done once every 160 sample frame while LSPs are updated every 2.5ms. The gain adjustment factor r_{adj} is computed as follows:

$$r_{adj} = \sqrt{\frac{\sum_{n=0}^{159} \{s(n)\}^2}{\sum_{n=0}^{159} \{s_{Pf}(n)\}^2}}$$

The spectrally shaped and gain adjusted output $s_{pf}'(n)$ is obtained as:

$$s_{Pf}'(n) = r_{adj} s_{Pf}(n) \quad (0 \le n \le 159)$$

Smoothing process:

The spectrally shaped and gain adjusted output $s_{Pf}'(n)$ is then smoothed to avoid discontinuities due to parameter change at the beginning of each frame. Postfiltered voiced output $vspeech(n)$ is obtained as follows

$$vspeech(n) = \begin{cases} \left(1 - \frac{n}{20}\right) s_{Pf\_prev}'(n) + \frac{n}{20} s_{Pf}'(n) & (0 \le n \le 19) \\ s_{Pf}'(n) & (20 \le n \le 159) \end{cases}$$

where $s_{Pf\_prev}'(n)$ is a $s_{Pf}'(n)$ computed using previous frame's α_n and r_{adj}.

2.B.1.3.2 Unvoiced speech

Spectral shaping:

$$Pf_{uv}(z) = \frac{\sum_{n=0}^{P} \alpha_n \gamma^n z^{-n}}{\sum_{n=0}^{P} \alpha_n \beta^n z^{-n}} \left(1 - \delta z^{-1}\right)$$

Similarly the transfer function of the spectral shaping filter for unvoiced speech is given as $Pf_{uv}(z)$, where α_n are linear predictive coefficients converted from de-quantized LSPs, which are updated every 20 ms. When normal decode mode is selected, LSP coefficients are updated at the middle of decode frame interval. If low delay decode mode is selected , decode frame interval is shifted by 2.5 ms and LSP update happens at 7.5 ms point from the beginning of the decode interval in Figure 2.9 $p = 10, \gamma = 0.5, \beta = 0.8, \delta = 0.1$. The **gain adjustment** and the **smoothing process** are the same as in the voiced part described above, and produces postfiltered unvoiced speech $uvspeech(n)$.

The output of each of the postfilters, $vspeech(n)$ and $uvspeech(n)$ are added to generate postfiltered speech output.

2.B.2 Post processing

2.B.2.1 Tool description

The output of the postfilter is fed into the post processing part. Post processing is composed of three filters, those are, high pass filter, high frequency emphasis filter, and low pass filter. High pass filter is used to remove unnecessary low frequency components, high frequency emphasis is used to increase the brightness of the speech, and low pass filter is used to remove unnecessary high frequency components. The filter configurations and constants described here is one example and not normative, and they could be modified.

2.B.2.2 Definitions

$HPF(z)$: Transfer function of high pass filter.

$Emp(z)$: Transfer function of high frequency emphasis filter.

$LPF(z)$: Transfer function of low pass filter.

2.B.2.3 Processing

The three filters below are applied to the output of the postfilter.

High Pass Filter:

$$HPF(z) = G_{inv} \frac{1 + A_1 z^{-1} + B_1 z^{-2}}{1 + C_1 z^{-1} + D_1 z^{-2}} \cdot \frac{1 + A_2 z^{-1} + B_2 z^{-2}}{1 + C_2 z^{-1} + D_2 z^{-2}}$$

High Frequency Emphasis:

$$Emp(z) = GG\frac{1 + AA\ z^{-1} + BB\ z^{-2}}{1 + CC\ z^{-1} + DD\ z^{-2}}$$

Low Pass Filter:

$$LPF(z) = GL\frac{1 + AL\ z^{-1} + BL\ z^{-2}}{1 + CL\ z^{-1} + DL\ z^{-2}}$$

2.B.2.4 Tables

Table 2.B.1 — Coefficients of the high pass filter

G_{inv}	1.100000000000000
A_1	-1.998066423746901
B_1	1.000000000000000
C_1	-1.962822436245804
D_1	0.9684991816600951
A_2	-1.999633313803449
B_2	0.9999999999999999
C_2	-1.858097918647416
D_2	0.8654599838007603

Table 2.B.2 — Coefficients of the high frequency emphasis filter

AA	0.551543
BB	0.152100
CC	0.89
DD	0.198025
GG	1.226

Table 2.B.3 — Coefficients of the low pass filter

AL	-2. * 1. * cos((4.0/4.0)* π)
BL	1.
CL	-2. * 0.78 * cos((3.55/4.0)* π)
DL	0.78*0.78
GL	0.768

Annex 2.C
(informative)

System layer definitions

2.C.1 Random access point

Random access point in HVXC bitstream can be set at any frame boundary point.

Annex 2.D
(informative)

Example of EP tool setting and error concealment for HVXC

2.D.1 Overview

This section describes one example of the implementation of EP (Error Protection) tool and error concealment method for HVXC. Some of perceptually important bits are protected by FEC (forward error correction) scheme and some are checked by CRC to judge whether or not erroneous bits are included. When CRC error occurs, error concealment is executed to reduce perceptible degradation.

It should be noted that error correction method and EP tool setting, error concealment algorithm described below are one example, and they should be modified depending on the actual channel conditions.

2.D.2 EP tool setting

2.D.2.1 Out-band Information Example for HVXC

- 2 kbit/s fixed rate

ESC(Error Sensitivity Category) instances of two adjacent frames are applied for EP classes directly:
Class 1: 22 bit (fixed), 2frame concatenated, SRCPC code rate 8/16, 6 bit CRC
Class 2: 4 bit (fixed), SRCPC code rate 8/8,1 bit CRC
Class 3: 4 bit (fixed), SRCPC code rate 8/8,1 bit CRC
Class 4: 20 bit (fixed), 2frame concatenated, SRCPC code rate 8/8,no CRC

```
1                   /* number of predefined sets */
2                   /* bit interleaving */
0                   /* bitstuffing */
2                   /* 2 frame concatenate */
4                   /* number of classes */
0 0 0 1 0 0 2 0     /* length_esc, srcpc_esc, crc_esc, concatenate, FEC type, No termination, interleave SW, class option */
22                  /* bits used for class length (0 = until the end) */
8                   /* puncture rate for srcpc 0 = 8/8 ... 24 = 32/8 */
6                   /* crc length */
0 0 0 0 0 0 3 0     /* length_esc, srcpc_esc, crc_esc, concatenate, FEC type, No termination, interleave SW, class option */
4                   /* bits used for class length (0 = until the end) */
0                   /* puncture rate for srcpc 0 = 8/8 ... 24 = 32/8 */
1                   /* crc length */
0 0 0 0 0 0 3 0     /* length_esc, srcpc_esc, crc_esc, concatenate, FEC type, No termination, interleave SW, class option */
4                   /* bits used for class length (0 = until the end) */
0                   /* puncture rate for srcpc 0 = 8/8 ... 24 = 32/8 */
1                   /* crc length */
0 0 0 1 0 0 3 0     /* length_esc, srcpc_esc, crc_esc, concatenate, FEC type, No termination, interleave SW, class option */
```

10	/* bits used for class length (0 = until the end) */
0	/* puncture rate for srcpc 0 = 8/8 ... 24 = 32/8 */
0	/* crc length */

- 4 kbit/s fixed rate

ESC(Error Sensitivity Category) instances of two adjacent frames are applied for EP classes directly:
Class 1: 33 bit (fixed), 2frame concatenated, SRCPC code rate 8/16, 6bit CRC
Class 2: 22 bit (fixed), 2frame concatenated, SRCPC code rate 8/8, 6bit CRC
Class 3: 4 bit (fixed), SRCPC code rate 8/8, 1bit CRC
Class 4: 4 bit (fixed), SRCPC code rate 8/8, 1bit CRC
Class 5: 17 bit (fixed), 2frame concatenated, SRCPC code rate 8/8, no CRC

1	/* number of predefined sets */
2	/* 1 bit interleaving */
0	/* bitstuffing */
2	/* 2 frame concatenate */
5	/* number of classes */
0 0 0 1 0 0 2 0	/* length_esc, srcpc_esc, crc_esc, concatenate, FEC type, No termination, interleave SW, class option */
33	/* bits used for class length (0 = until the end) */
8	/* puncture rate for srcpc 0 = 8/8 ... 24 = 32/8 */
6	/* crc length */
0 0 0 1 0 0 3 0	/* length_esc, srcpc_esc, crc_esc, concatenate, FEC type, No termination, interleave SW, class option */
22	/* bits used for class length (0 = until the end) */
0	/* puncture rate for srcpc 0 = 8/8 ... 24 = 32/8 */
6	/* crc length */
0 0 0 0 0 0 3 0	/* length_esc, srcpc_esc, crc_esc, concatenate, FEC type, No termination, interleave SW, class option */
4	/* bits used for class length (0 = until the end) */
0	/* puncture rate for srcpc 0 = 8/8 ... 24 = 32/8 */
1	/* crc length */
0 0 0 0 0 0 3 0	/* length_esc, srcpc_esc, crc_esc, concatenate, FEC type, No termination, interleave SW, class option */
4	/* bits used for class length (0 = until the end) */
0	/* puncture rate for srcpc 0 = 8/8 ... 24 = 32/8 */
0	/* crc length */
0 0 0 1 0 0 3 0	/* length_esc, srcpc_esc, crc_esc , concatenate, FEC type, No termination, interleave SW, class option */
17	/* bits used for class length (0 = until the end) */
0	/* puncture rate for srcpc 0 = 8/8 ... 24 = 32/8 */
0	/* crc length */

The Table 2.D.1 below shows a channel coded bit assignment for the use of the above EP tool settings.

Table 2.D.1 — The channel coded bit assignment for the use of the EP tool

	2 kbit/s fixed rate	4 kbit/s fixed rate
Class I		
Source coder bits	44 (*1)	66 (*1)
CRC parity	6	6
Code Rate	8/16	8/16
Class I total	100	144
Class II		
Source coder bits	4	44 (*1)
CRC parity	1	6
Code Rate	8/8	8/8
Class II total	5	50
Class III		
Source coder bits	4	4
CRC parity	1	1
Code Rate	8/8	8/8
Class III total	5	5
Class IV		
Source coder bits	4	4
CRC parity	1	1
Code Rate	8/8	8/8
Class IV total	5	5
Class V		
Source coder bits	4	4
CRC parity	1	1
Code Rate	8/8	8/8
Class V total	5	5
Class VI		
Source coder bits	20 (*1)	4
CRC parity	0	1
Code Rate	8/8	8/8
Class VI total	20	5
Class VII		
Source coder bits		34 (*1)
CRC parity		0
Code Rate		8/8
Class VII total		34
Total Bit of All Classes	140	248
Bitrate	3.5 kbit/s	6.2 kbit/s

(*1) 2 frame concatenated.

Class I:

CRC covers all the Class I bits, and Class I bits including CRC are protected by convolutional coding.

Class II-V(2 kbit/s), II-VI(4 kbit/s):

At least one CRC bits cover the source coder bits of these classes.

Class VI(2 kbit/s), VII(4 kbit/s) :

The source coder bits are not checked by CRC nor protected by any error correction scheme

2.D.3 Error concealment

When CRC error is detected, error concealment processing (bad frame masking) is carried out. An example of concealment method is described below.

A frame masking state of the current frame is updated based on the decoded CRC result of Class I. The state transition diagram is shown in Figure 2.D.1. The initial state is state = 0. The arrow with a letter “1” denotes the transition for a bad frame, and that with a letter “0” a good frame.

2.D.3.1 Parameter replacement

According to the state value, the following parameter replacement is done. In error free condition, state value becomes 0, and received source coder bits are used without any concealment processing.

2.D.3.1.1 LSP parameters

At state = 1..6, LSP parameters are replaced with those of previous ones.

When state = 7, If LSP4 = 0 (LSP quantization mode without inter-frame prediction), then LSP parameters are calculated from all LSP indices received in the current frame. If LSP4 = 1 (LSP quantization mode with inter-frame coding), then LSP parameters are calculated with the following method.

In this mode, LSP parameters from LSP1 index are interpolated with the previous LSPs.

$$LSP_{base}(n) = p \cdot LSP_{prev}(n) + (1-p)LSP_{1st}(n) \qquad \text{for n = 1..10} \quad (1)$$

$LSP_{base}(n)$ is LSP parameters of the base layer, $LSP_{prev}(n)$ is the previous LSPs, $LSP_{1st}(n)$ is the decoded LSPs from the current LSP1 index, and p is the factor of interpolation. p is changed according to the number of previous CRC error frames of Class I bits as shown in Table 2.D.2. LSP indices LSP2, LSP3 and LSP5 are not used and $LSP_{base}(n)$ is used as current LSP parameters.

Table 2.D.2 — p factor

Frame	p
0	0.7
1	0.6
2	0.5
3	0.4
4	0.3
5	0.2
6	0.1
≥7	0.0

2.D.3.1.2 Mute variable

According to the “state” value, a variable “mute” is set to control output level of speech.

The “mute” value below is used.

In state = 7, the average of 1.0 and “mute” value of the previous frame(= 0.5 (1.0 + previous “mute value”)) is used, but when this value is more than 0.8, “mute” value is replaced with 0.8.

Table 2.D.3 — mute variable

State	mute
0	1.000
1	0.800
2	0.700
3	0.500
4	0.250
5	0.125
6	0.000
7	Average/0.800

2.D.3.1.3 Replacement and gain control of “voiced” parameters

In state = 1..6, spectrum parameter SE_shape1, SE_shape2, spectrum gain parameter SE_gain, spectrum parameter for 4 kbit/s codec SE_shape3 .. SE_shape6 are replaced with corresponding parameters of the previous frame. Also, to control volume of output speech, harmonic magnitude parameters of LPC residual signal “$Am[0\ldots127]$” is gain controlled as shown in Eq.(1). In the equation, $Am_{(org)}[i]$ is computed from the received spectrum parameters from the latest error free frame.

$$Am[i] = mute * Am_{(org)}[i] \quad \text{for i=0..127} \qquad (1)$$

If previous frame is unvoiced and current state is state=7, Eq.(1) is replaced with Eq.(2).

$$Am[i] = 0.6 * mute * Am_{(org)}[i] \quad \text{for i=0..127} \qquad (2)$$

As described before, SE_shape1 and SE_shape2 are individually protected by 1 bit CRC. In state = 0 or 7, when CRC errors of these classes are detected at the same time, the quantized harmonic magnitudes with fixed dimension $Am_{qnt}[1..44]$ are gain suppressed as shown in Eq.(3).

$$Am_{qnt}[i] = s[i] * Am_{qnt(org)}[i] \quad \text{for i=1..44} \qquad (3)$$

$s[i]$ is the factor for the gain suppression.

Table 2.D.4 — factor for gain suppression ‘s[0..44]’

i	1	2	3	4	5	6	7..44
$s[i]$	0.10	0.25	0.40	0.55	0.70	0.85	1.00

At 4 kbit/s, SE_shape4, SE_shape5, and SE_shape6 are checked by CRC as Class II bits. When CRC error is detected, the spectrum parameter of the enhancement layer is not used.

2.D.3.1.4 Replacement and gain control of "unvoiced" parameters.

In state = 1..6, stochastic codebook gain parameter VX_gain1[0], VX_gain1[1] are replaced with the VX_gain1[1] from the latest error free frame. Also stochastic codebook gain parameter for 4 kbit/s codec VX_gain2[0]..VX_gain2[3] are replaced with the VX_gain2[3] from the latest error free frame.

Stochastic codebook shape parameter VX_shape1[0], VX_shape1[1],and stochastic codebook shape parameter for 4 kbit/s codec are generated from randomly generated index values.

Also, to control volume of output speech, LPC residual signal $res[0\ldots159]$ is gain controlled as shown in Eq.(4). In the equation, $res_{(org)}[i]$ is computed from stochastic codebook parameters.

$$res[i] = mute * res_{(org)}[i] \qquad (0 \le i \le 159) \qquad (4)$$

2.D.3.1.5 Frame Masking State Transitions

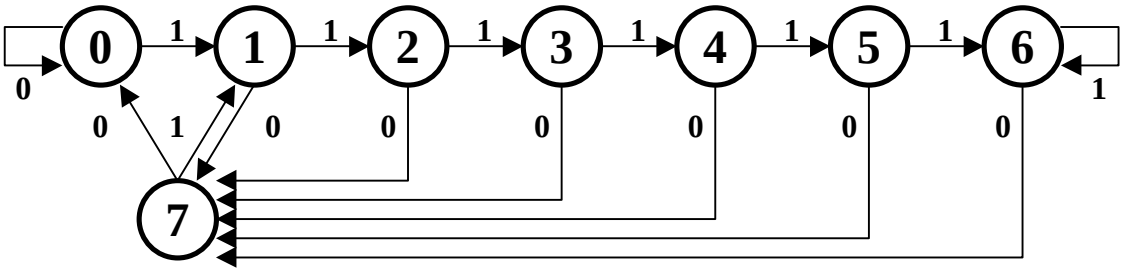


Figure 2.D.1 — Frame Masking State Transitions

Annex 2.E
(normative)

VQ codebooks for HVXC

2.E.1 List of the VQ codebooks

In this Annex, VQ codebook tables listed in the Table 2.E.1 are given. All the codeword values in the tables in this Annex must be divided by the values indicated as a factor for each of the tables before use.

Table 2.E.1 — List of the VQ codebooks

harmonic VQ table - 2k	**CbAm**
harmonic VQ table - 4k	**CbAm4k**
stochastic codebook table - 2k	**CbCelp**
stochastic codebook table - 4k	**CbCelp4k**
LSP quantizer table - 2k	**CbLsp**
LSP quantizer table - 4k	**CbLsp4k**

2.E.2 CbAm

VQ codebook for harmonic spectral vector for 2 kbps

Table 2.E.2 — Spectral gain codebook (base layer): g0[]
dim= 1 x 32 codewords
16 bits signed
factor = 2^2

index	Codeword	index	codeword
0	**57**	**16**	**26628**
1	**98**	**17**	**22212**
2	**206**	**18**	**17853**
3	**152**	**19**	**20138**
4	**625**	**20**	**12642**
5	**480**	**21**	**13691**
6	**278**	**22**	**16103**
7	**364**	**23**	**14774**
8	**4544**	**24**	**5362**
9	**3787**	**25**	**6172**
10	**2460**	**26**	**8005**
11	**3077**	**27**	**7125**
12	**826**	**28**	**11519**
13	**1110**	**29**	**10688**
14	**1940**	**30**	**8954**
15	**1482**	**31**	**9856**

Table 2.E.3 — Spectral shape (base layer): cb0[16][44]
dim = 44 * 16 codevectors
16 bits signed
factor =2^17

index (SE_shape1)	codeword			
0	7710	12303	6176	1720
	5594	7400	2762	185
	2501	4264	3853	2958
	4090	6822	6583	4672
	3971	4733	3289	2402
	3414	4836	5483	5375
	5788	6150	6913	5797
	4675	4611	5268	3911
	2219	1691	2596	3258
	4269	5670	7460	8297
	6711	3040	2429	938
1	6406	10409	8027	7324
	9265	7299	2691	-89
	547	2846	4081	3718
	2848	4351	5958	8480
	9650	8783	5877	4566
	5500	7227	8808	8891
	7361	6209	5672	5526
	5247	5120	5262	3651
	1414	881	2038	3277
	3965	4049	4623	5491
	5508	6985	12000	11198
2	3235	5535	5477	5511
	6480	6626	5549	3676
	2540	3077	4530	4482
	3887	6011	6418	6445
	6392	7353	6602	4748
	4929	6353	6300	4968
	4259	4319	4881	6797
	11052	13332	15290	16039
	13107	9461	5263	2697
	2047	2080	3535	4610
	3164	-679	-831	700
3	4236	9106	11295	11039
	10359	9003	6981	4421
	2691	2005	2262	2279
	2289	4095	4052	3312
	3592	3944	2875	1864
	2101	3171	3253	3162
	2987	3429	3525	3078
	2896	2494	1813	920
	292	715	2401	3234
	4049	4442	4998	5256
	3711	-395	-154	1377
4	13954	8825	-2191	957
	2890	2461	2722	2351
	2592	3760	4872	3592
	1614	3113	3932	5058
	5208	4689	2860	1190
	1548	3179	4830	4917

	5052	5547	5793	5944
	6135	5499	3854	2075
	473	1034	2835	3291
	4368	5192	6498	6736
	4248	536	-645	-861
	2225	5420	8507	10359
	9443	4811	-1364	-4304
	-2831	434	2964	3312
	2106	2885	2677	2142
	2840	4028	3783	2347
5	2311	2970	3318	2793
	1998	2021	2407	2170
	1991	2124	2534	2117
	1596	1733	1779	1550
	1795	1555	2944	4183
	2887	-736	-1242	-152
	1326	4724	9967	15539
	16304	10623	3163	-788
	1521	5819	8273	6952
	4371	4060	3718	3298
	4107	5364	4891	3022
6	3069	4169	4688	3447
	2611	3101	3915	4441
	4701	4975	3489	1533
	-17	219	1874	3136
	4116	3947	4988	5386
	3392	-1057	-2291	-1434
	5676	11275	11435	8100
	4416	1903	429	-130
	1590	4552	7473	7119
	4485	3794	2766	2043
	2936	3365	2673	1994
7	2694	4167	4494	4121
	3967	4379	4656	4373
	3685	2961	2096	1185
	552	1424	3723	5183
	5873	5071	5288	5628
	4194	397	-662	25
	8006	14433	13555	7820
	5616	9749	15468	13912
	8212	4869	5584	7375
	9117	12166	11620	9748
	9115	9020	8153	7046
8	7170	8236	8790	8369
	8279	8558	9550	9980
	9885	8427	7520	5986
	5138	5211	7231	9033
	10498	11176	12207	12670
	10706	7004	6441	4155
	5036	10205	11528	9840
	9263	8782	7973	7590
	7427	7360	7549	6667
	6528	9201	10278	10508
	11114	10054	7920	6415
9	6869	8687	10127	11499
	13521	16759	17481	14237
	9991	7277	5499	4023

	3069	3202	4544	5645
	7247	9846	14884	22076
	23893	17146	9262	2938
10	2928	6339	7501	7908
	8465	8757	7683	6864
	6934	7277	8001	6882
	4939	6921	8146	8670
	9110	9483	8002	6317
	6217	6919	6816	6580
	6868	7313	7860	7728
	7708	7120	4417	2995
	5624	13956	22866	26244
	25377	20595	14318	8088
	3734	-843	-1401	-767
11	2324	4896	6745	9373
	13292	15734	15021	11516
	8153	6101	5412	4278
	3763	6757	9367	11234
	12719	13716	12301	10678
	10558	11499	11287	9989
	8202	7439	7656	8212
	8505	8095	7800	6342
	5458	5804	7203	8282
	8899	9103	10506	11557
	10899	8015	7049	5679
12	5291	9095	6217	2989
	3371	6555	9455	10807
	10012	8879	8801	7892
	6204	8161	9912	10641
	9898	9877	8923	8066
	7580	7453	7666	7519
	7339	8115	8627	8928
	8720	7023	5735	4203
	3283	3492	4904	5389
	4974	4628	5865	7185
	9450	14948	24101	24286
13	7485	7194	760	6246
	12284	4764	-576	2634
	5063	4651	3470	2519
	2409	4105	3939	3596
	4844	5418	4188	3338
	3544	4956	5411	5607
	5167	5010	5619	5694
	4935	4568	3990	2538
	744	412	2351	3498
	5146	6572	8301	8349
	6779	2113	665	1038
14	3861	7854	10553	12074
	13784	13715	11321	7859
	6536	8311	12485	16123
	17146	16537	12534	9378
	8763	9137	8451	7949
	8435	9215	9124	8284
	7246	6931	7589	8877
	10426	10428	10085	8625
	7294	7266	8688	9873
	10573	10885	12186	13044

	11178	6062	4333	3668
15	5319	10458	11442	9126
	6605	4864	4486	5813
	8369	9902	9479	7280
	5596	7771	9447	10339
	12120	13645	13169	11839
	11237	10432	8377	6807
	5851	5981	6977	8020
	10188	11005	9799	7195
	4792	4291	5736	6737
	6776	6560	7849	9576
	8904	5430	4240	2797

Table 2.E.4 — Spectral shape (base layer): cb1[16][44]
dim=44 * 16 codevectors
16 bits signed
factor = 2^17

index (SE_shape2)	Codeword			
0	5073	9567	9861	7804
	9063	9510	6902	4357
	3660	4111	4709	6181
	7125	6090	8049	9594
	7528	5771	6747	7965
	7779	6443	5107	5658
	6264	6812	7159	7779
	9055	8740	7823	8271
	9104	9062	7726	6345
	5475	5241	4112	4086
	6450	10299	10485	7481
1	7272	13230	10938	6723
	11197	18250	15132	9622
	9201	10191	10757	10474
	10315	8562	8277	9264
	9009	8390	8987	9757
	9670	9058	9393	10433
	11379	10843	9726	9623
	9375	9673	9576	10383
	11793	12855	14061	14168
	13653	14983	15013	13399
	12238	13728	11364	6705
2	2959	7446	13747	18610
	18985	17157	15474	13007
	9416	6379	4671	5822
	9248	10374	11050	10787
	9149	8527	9056	9764
	8961	7911	8203	9124
	8923	8179	7701	7757
	8100	8968	10546	12239
	13814	13619	11804	9965
	8487	7732	6518	5977
	7981	11564	10534	7579
	881	3834	9534	15687
	19571	21127	21317	20686
	18354	16020	14101	14310

	15415	13359	12665	12287
	11855	11085	11885	12729
3	12010	11130	10545	11096
	12206	12018	11505	11086
	10859	11481	12562	14110
	16114	16406	14734	13185
	12023	11632	10977	11301
	14069	18042	15926	11411
	7360	14622	15669	9368
	2755	2083	5117	7701
	7120	4817	3060	3642
	4554	3001	2864	3714
	4100	3934	4084	4683
4	4396	3489	2996	3929
	5090	5027	4643	4134
	3751	3700	4283	5958
	7244	7072	5787	4815
	4386	3571	1939	1097
	2929	6701	7463	7336
	5196	10635	12117	10303
	7333	5421	5336	6648
	9173	12792	15868	15774
	11876	5957	4484	5594
	6152	5943	7013	8206
5	7946	6912	6904	8086
	9500	9693	9573	9461
	8868	9403	9563	10010
	10884	10854	9298	7607
	7178	7287	6657	5052
	6361	10084	8921	5501
	3814	7967	12276	15441
	13000	8461	5054	3396
	3144	3838	5477	7326
	8611	6356	5259	4945
	4360	4073	5227	6583
6	6294	5237	5170	5842
	6016	5587	5788	5732
	5040	4746	5318	6409
	7897	7905	6350	5237
	4578	4646	4151	4475
	8546	16193	17112	11921
	3614	8494	13158	15420
	14353	11638	9963	11367
	14001	15703	14012	11770
	10243	7114	6543	7277
	7620	7790	9649	11153
7	10989	9370	8266	7678
	8114	8703	9417	10506
	10421	10014	9189	10121
	11809	12712	12751	12325
	10815	9566	8154	8222
	9942	12697	10582	6937
	9640	16893	13829	7074
	5462	9678	15386	15346
	10141	7792	9380	12736
	13929	10195	8097	8979
	9992	9693	10282	11185

8	11110	9595	8741	8421
	8601	8993	9737	9894
	10042	10655	11404	12775
	14831	15267	13130	11306
	10232	9384	8076	7109
	8837	12760	11663	6908
9	4431	9140	11976	11846
	10802	10726	11703	12710
	11853	10104	8574	9669
	11691	10610	10325	10593
	9775	9589	11402	14198
	15648	15642	15299	14194
	11693	8709	6792	6841
	7525	9517	12122	16429
	20201	21244	18251	13723
	8741	6187	5201	5710
	8559	12380	10814	7065
10	6417	13791	17968	17891
	13116	9753	9387	10587
	10112	9537	8292	9676
	11265	10567	10373	10581
	10035	9550	10655	11942
	11224	10169	9913	10739
	12516	13678	13764	13060
	11483	10871	11190	10454
	11006	11958	13540	17694
	22764	25085	23894	17793
	12022	10893	8946	5785
11	6643	13495	16708	14546
	9833	8439	12580	19060
	21155	17246	11261	8975
	10488	10761	11801	12682
	12643	11747	12055	12582
	12365	11183	10826	11197
	10922	10288	9831	10217
	10597	11134	11651	13620
	15546	15759	14419	13073
	11822	11093	10186	10018
	11946	15324	13076	8063
12	9320	13367	7272	3508
	4787	6638	6347	4309
	3669	4306	5158	5702
	6564	4750	4151	4588
	4773	4885	6161	6830
	6057	4794	4497	5135
	5719	5897	5270	4610
	4953	6288	6932	7984
	8446	7828	6074	5357
	4867	4268	2723	3372
	7136	12675	11294	5078
13	9271	12500	5475	5105
	13647	13655	7248	5595
	5206	5703	8553	11643
	10651	7301	6630	8221
	7870	7294	8693	9878
	9740	8182	8139	7921
	8812	9644	10152	10275

	10590	10528	9952	9091
	9880	10394	8606	7573
	6580	5574	3725	2508
	5452	13958	21519	22685
14	7366	14917	18246	15291
	9380	6686	7609	8991
	8434	6401	4936	7007
	11702	13856	13723	12350
	10270	8661	9110	10490
	10160	8741	8462	8722
	8443	8320	8454	8981
	9163	10173	10973	11715
	12739	12526	10981	9192
	7477	6617	5552	5244
	7887	14437	15124	10911
15	3647	8297	13029	15393
	13816	10428	7568	6298
	5446	5598	6989	10927
	14126	13634	13227	13814
	13809	13528	12929	11754
	9866	8138	8304	9576
	11156	12009	11476	10823
	10434	10355	11424	13811
	16041	16689	15293	12292
	8551	5943	4131	4170
	7446	13011	12150	8225

2.E.3 CbAm4k

VQ codebook for harmonic spectral vector quantization for 4kbps.

Table 2.E.5 — Spectral shape (enhabcement layer): cb4k[0][128][2]
dim = 2x128 codewords
16 bits signed
factor = 2^6

index (SE_shape3)	codeword	
0	-5773	858
1	-1768	1201
2	644	1969
3	444	46
4	-2723	3150
5	-121	172
6	-4459	6812
7	-1193	2298
8	-11663	2137
9	-2386	-178
10	-3323	1768
11	-195	-15
12	-2449	567
13	45	-11
14	-7228	3785
15	-2010	1917
16	-1169	3581

17	417	284
18	-684	8757
19	-439	2544
20	-2127	14132
21	501	4365
22	-3449	25883
23	-3584	9653
24	-3401	917
25	-172	-463
26	-848	6067
27	380	1285
28	-4159	3823
29	-1099	810
30	-4328	16051
31	-2563	4871
32	970	382
33	388	-200
34	7754	1041
35	2254	1818
36	698	598
37	745	-435
38	1684	2655
39	842	137
40	-4839	2005
41	-50	627
42	1287	-141
43	701	-127
44	-169	1056
45	242	-411
46	-834	1453
47	18	93
48	4585	2897
49	1556	348
50	9446	6193
51	3441	976
52	2890	4293
53	1702	898
54	5292	11266
55	274	3043
56	305	827
57	528	-791
58	4109	29
59	1634	-459
60	-210	1695
61	-83	-244
62	2193	7152
63	1071	1264
64	5773	-858
65	1768	-1201
66	-644	-1969
67	-444	-46
68	2723	-3150
69	121	-172
70	4459	-6812
71	1193	-2298
72	11663	-2137
73	2386	178

74	3323	-1768
75	195	15
76	2449	-567
77	-45	11
78	7228	-3785
79	2010	-1917
80	1169	-3581
81	-417	-284
82	684	-8757
83	439	-2544
84	2127	-14132
85	-501	-4365
86	3449	-25883
87	3584	-9653
88	3401	-917
89	172	463
90	848	-6067
91	-380	-1285
92	4159	-3823
93	1099	-810
94	4328	-16051
95	2563	-4871
96	-970	-382
97	-388	200
98	-7754	-1041
99	-2254	-1818
100	-698	-598
101	-745	435
102	-1684	-2655
103	-842	-137
104	4839	-2005
105	50	-627
106	-1287	141
107	-701	127
108	169	-1056
109	-242	411
110	834	-1453
111	-18	-93
112	-4585	-2897
113	-1556	-348
114	-9446	-6193
115	-3441	-976
116	-2890	-4293
117	-1702	-898
118	-5292	-11266
119	-274	-3043
120	-305	-827
121	-528	791
122	-4109	-29
123	-1634	459
124	210	-1695
125	83	244
126	-2193	-7152
127	-1071	-1264

Table 2.E.6 — Spectral shape (enhabcement layer): cb4k[1][1024][4]
dim = 4x1024 codewords
16 bits signed
factor = 2^6

index (SE_shape4)	codeword			
0	**-3708**	**4595**	**174**	**5466**
1	**-6642**	**6292**	**-2437**	**-442**
2	**-5553**	**3819**	**1853**	**8413**
3	**-425**	**627**	**-5619**	**3556**
4	**-7817**	**7075**	**-10600**	**2258**
5	**-3925**	**2833**	**72**	**1358**
6	**-11037**	**7197**	**-5288**	**14434**
7	**-2425**	**4350**	**-333**	**9257**
8	**-1515**	**-215**	**194**	**-90**
9	**-1555**	**2325**	**-2488**	**2620**
10	**-687**	**657**	**-3195**	**650**
11	**-3784**	**3980**	**-12574**	**-5902**
12	**-1194**	**1024**	**-2115**	**25**
13	**4326**	**4919**	**7640**	**-326**
14	**-793**	**2014**	**-1672**	**6939**
15	**-653**	**754**	**-2286**	**-318**
16	**-5673**	**1116**	**-8246**	**2246**
17	**-11688**	**8435**	**-16555**	**-2551**
18	**1247**	**1855**	**-4749**	**1838**
19	**1005**	**7635**	**-17838**	**475**
20	**-1301**	**1082**	**-2757**	**4124**
21	**-9574**	**1676**	**-8532**	**952**
22	**-2056**	**2257**	**-7359**	**4384**
23	**-2787**	**1039**	**-5899**	**11**
24	**-1070**	**4111**	**-1342**	**-1460**
25	**-1116**	**12475**	**-9995**	**-650**
26	**-1723**	**6725**	**-6020**	**-5427**
27	**1257**	**12863**	**-20202**	**-9518**
28	**-730**	**-2201**	**2676**	**-568**
29	**-426**	**2249**	**-2616**	**2826**
30	**-236**	**197**	**-921**	**-2407**
31	**-347**	**3829**	**-8218**	**-8722**
32	**-13058**	**10488**	**-8453**	**9753**
33	**-5056**	**2191**	**-4350**	**905**
34	**-1778**	**15731**	**-4754**	**10386**
35	**-2236**	**5103**	**-2860**	**3991**
36	**-7333**	**9303**	**-15598**	**16357**
37	**-6027**	**13542**	**-2855**	**7737**
38	**-61**	**12148**	**-12735**	**23167**
39	**1448**	**14926**	**-6518**	**13767**
40	**-8335**	**2500**	**1145**	**3713**
41	**-1633**	**-1142**	**-251**	**-2105**
42	**-3704**	**2818**	**-4206**	**5422**
43	**681**	**1046**	**-4699**	**-4321**
44	**-5111**	**8969**	**-10188**	**2045**
45	**-3247**	**1963**	**-1582**	**2605**
46	**-2955**	**7990**	**-14298**	**5719**
47	**-1580**	**4537**	**-825**	**6925**
48	**-4958**	**5233**	**-573**	**2004**
49	**-4890**	**14378**	**-10816**	**-4353**

50	-2741	2600	-3685	4183
51	-5465	1216	-8887	-3905
52	-6944	5380	-5414	7765
53	-3224	4618	-1955	1816
54	335	10495	-15036	6311
55	-90	4485	-1280	3781
56	290	-398	727	-2585
57	1072	3908	-3961	-8898
58	-1920	1344	-7820	-2107
59	4502	8827	-17295	-6552
60	-967	1650	1808	-4294
61	-480	493	-750	-1257
62	971	368	-5922	5544
63	-462	2029	-8023	-2102
64	-782	470	269	853
65	-661	803	-223	123
66	-2119	266	635	4444
67	-630	-1052	-315	942
68	-3596	473	-2921	1915
69	-277	-356	759	478
70	-2282	2421	-5053	10636
71	-1276	818	-1150	2733
72	355	-532	1715	-1003
73	-53	-1318	1661	582
74	632	-2107	-217	-1528
75	211	-551	-2387	-1385
76	-616	-292	501	527
77	-2509	364	5987	2601
78	391	166	-236	-3245
79	438	-1049	1580	-160
80	-5314	325	1110	295
81	-4272	653	-7224	3198
82	-1	-133	-365	268
83	-1713	1072	-5118	-100
84	-198	-287	48	211
85	-2646	-1197	-1365	624
86	-673	1509	-1153	1719
87	-236	-1532	-859	319
88	2237	58	5065	-7544
89	-1467	312	-3129	-1035
90	3474	-1701	-1247	-1553
91	7069	813	-11959	-7397
92	-61	-242	58	-2288
93	-588	131	1017	1208
94	3938	4608	-446	-10269
95	873	54	-2588	-3892
96	-7524	2222	-2307	5325
97	-912	628	-939	696
98	3404	1697	-1102	10570
99	-1136	-638	-1377	3107
100	1671	2529	-3694	9330
101	-697	3208	1806	5118
102	3035	3843	-15529	23374
103	-3701	2884	-9284	8114
104	-1406	-309	-1186	730
105	62	-62	8	32
106	1637	-1305	-1739	2757

107	274	18	-1460	41
108	-2344	265	-1672	629
109	-1109	3991	1669	2016
110	410	923	-10336	6935
111	-403	148	-1632	3694
112	-1298	-807	252	1014
113	325	3999	-3187	-3229
114	1073	1368	-2383	4904
115	-1227	255	-1438	-666
116	-670	14	-2684	3137
117	-287	103	63	135
118	255	3490	-11611	9597
119	522	36	-2783	4220
120	1835	-2453	1409	-1043
121	218	-1024	-813	-2278
122	-54	186	-408	-884
123	3969	4248	-5097	-4668
124	26	-991	1669	-862
125	-19	-137	245	-54
126	1085	6265	-376	1538
127	-108	-130	-1726	-1060
128	-1121	635	-236	988
129	-1737	1161	771	1046
130	-1419	1019	-760	1417
131	-356	144	-410	270
132	-4517	438	-722	1664
133	-763	3156	5529	9680
134	-2411	1731	1934	8028
135	1668	-1040	1629	2868
136	-6824	4650	-315	-2845
137	-810	3890	5416	3951
138	-1088	-249	-388	-310
139	-1714	-705	-2398	-520
140	1243	8345	6597	302
141	595	12918	13592	9238
142	-15	-744	809	-3756
143	2603	2817	8832	702
144	-3589	2930	274	-1360
145	-11450	7157	-1170	-474
146	-486	138	-202	-726
147	-198	3273	-5179	2160
148	-649	-2464	410	1835
149	-3013	-221	-1093	2541
150	360	458	-597	-1915
151	-129	75	-75	152
152	-5599	10181	-1527	-13002
153	-5539	2837	-1770	-1665
154	-2093	1851	-4449	-5297
155	-34	7568	-8611	-3506
156	-958	2360	-1516	-4489
157	1209	2368	7398	1540
158	6201	5468	-1745	-7751
159	1834	204	-3621	-1843
160	-4192	2003	-1263	9486
161	48	101	-1465	1441
162	-1987	5262	1146	4680
163	339	445	-67	1228

164	-9438	9847	-3534	4707
165	-1770	2859	992	3176
166	6020	10262	-774	10647
167	-12	1028	-1248	10011
168	-2051	-420	-50	956
169	652	-684	1852	1829
170	-93	186	-430	1151
171	94	-261	-2394	2313
172	-2006	504	-1688	1855
173	213	1148	4752	7458
174	-814	230	-4472	6898
175	-165	95	-3559	2178
176	-1963	-2056	207	2212
177	-2769	5330	-1410	-1443
178	25	331	158	1048
179	-827	-650	-2209	622
180	-5239	33	-1597	9961
181	-972	-1168	-365	2043
182	-1400	1650	-4458	4091
183	1630	305	-3728	3573
184	-5091	1693	2809	-2600
185	-258	-485	-1732	264
186	580	-2735	-2014	2558
187	125	135	-11841	2969
188	-1858	-1064	-981	1674
189	-2773	74	-4578	1621
190	1280	3555	-3417	-2632
191	-2153	3308	-15960	2431
192	-4241	958	416	-865
193	-3128	-2675	-610	-858
194	1026	-2422	-211	-571
195	-228	-3161	1008	-2019
196	-20	-63	-54	-41
197	-986	-1518	1451	2295
198	9	1388	-29	-3287
199	-456	-443	-111	600
200	-2355	38	8709	-6915
201	-1567	-840	1919	-117
202	3597	48	-3335	-6013
203	1694	-2322	315	-1579
204	-180	1768	-1223	-4678
205	-5114	3261	8514	1950
206	6386	179	-1249	-16705
207	1122	813	-965	-5297
208	-10496	6615	-2131	-9755
209	-7880	1186	-407	393
210	-1599	20	-3765	-2851
211	-842	-123	476	-700
212	-900	493	-116	-3041
213	-1045	-407	-332	589
214	-747	5161	-3022	-10826
215	4573	1692	339	-870
216	-3881	3559	5075	-19570
217	-2652	1226	401	-4968
218	7038	1565	-1483	-12077
219	899	-918	-616	-2329
220	7936	3162	3507	-6532

221	3970	187	3887	1722
222	12721	12975	-942	-19702
223	12004	5659	2488	-4690
224	-352	266	-911	1398
225	-1089	-2159	-699	-243
226	2629	-586	-1150	2141
227	-129	-893	397	804
228	-2755	289	-1942	3435
229	-287	213	683	701
230	3615	4600	-6882	6219
231	-1395	23	-990	6559
232	-2218	-2573	606	-2765
233	-677	-3951	1530	86
234	130	-3203	-1032	-1364
235	-276	-3743	-919	457
236	-469	127	428	-45
237	-1603	387	1020	80
238	1750	1781	-1595	-3811
239	157	-203	-318	687
240	-2340	127	-2231	-2730
241	-526	67	38	658
242	2557	-1795	986	-113
243	680	-1638	-353	-266
244	-1100	13	-1727	1545
245	839	-2868	-84	593
246	-9	5786	-565	2024
247	-148	1	-83	655
248	-1465	226	-319	-5095
249	-1974	-2118	-1354	-1007
250	3150	2292	14	-2752
251	780	-2824	-2501	-139
252	958	5024	1323	252
253	11	-25	7	354
254	9236	13635	-293	-4887
255	1463	3887	-2164	1698
256	-708	3836	-6314	-3506
257	-1970	17471	-12464	-351
258	-869	208	-168	1846
259	-1071	5458	-2417	569
260	-1464	1135	-2463	1733
261	-1187	3095	-5791	1952
262	-4159	3332	-734	3910
263	2275	10821	395	4807
264	86	-624	-1147	-246
265	-1117	4502	-4199	-2764
266	166	52	-118	173
267	449	3424	-3234	696
268	-311	-1708	2741	-2083
269	-2785	4153	3701	2760
270	-1135	302	550	-1588
271	-1184	971	9	4240
272	-808	68	-1424	26
273	-5136	6187	-3183	-3564
274	-7	-188	-960	324
275	748	1252	-7581	1604
276	-14	-912	2205	-1789
277	-1405	751	-1377	554

278	-648	-514	-1042	836
279	143	953	-515	1550
280	-883	-725	612	-3834
281	-662	2368	-2651	1351
282	1726	4225	973	-1570
283	4675	12839	-9323	-2710
284	-3621	174	12424	-5116
285	-1433	-1693	1521	-2069
286	-267	1064	4460	-2164
287	-1065	3217	-2198	1709
288	-6483	74	-2695	5213
289	2023	3583	-5542	-1418
290	779	3571	-587	5003
291	8624	5214	-562	3138
292	-110	2848	-5952	6395
293	2292	3251	-2925	2462
294	1034	10647	-9901	8040
295	14588	11498	-1318	8601
296	413	2206	934	-3276
297	140	1706	-22	1385
298	206	1621	-209	604
299	5955	10295	72	-2099
300	19	1128	-1935	-7769
301	-1370	1144	2203	378
302	-2795	669	-3004	3822
303	7437	1868	-4126	905
304	-400	26	554	-3070
305	-3637	2283	-3713	-5056
306	-433	901	222	-712
307	-699	8416	770	-46
308	1113	2821	2692	-2759
309	-294	697	400	98
310	1524	1989	-3552	4000
311	5160	4928	2172	1541
312	-1064	1709	2875	-15351
313	-760	6388	-492	-3470
314	-2026	4468	-1561	-5057
315	11	19162	-6693	-6267
316	3885	2603	9824	-16992
317	985	1742	339	-6632
318	2117	350	1174	-5597
319	2698	11997	-4125	-5799
320	-232	304	-1121	142
321	-3770	4053	-6203	2270
322	-2	-2321	-597	1057
323	-451	681	-671	599
324	609	379	1198	1452
325	-3582	-989	1566	1552
326	531	1194	-1106	1851
327	1116	716	1651	2403
328	-81	-2175	1785	-429
329	-2400	529	1340	1498
330	3614	-1796	546	-359
331	1849	-1169	-37	562
332	-3540	-1102	2407	-391
333	-12764	592	7511	2433
334	-715	6994	3431	-7914

335	-2516	-1617	2003	-1330
336	-654	-705	256	57
337	-179	653	-1087	-743
338	3320	-318	676	253
339	-154	-473	-615	477
340	-735	-944	1342	-358
341	-503	-742	1803	1393
342	310	1132	4767	542
343	134	-4	288	12
344	1170	828	3958	-1735
345	-1877	636	180	724
346	14629	473	2203	-5314
347	4959	3220	-1851	716
348	-3032	2368	6663	-1038
349	-5137	1085	2171	6217
350	4773	7174	12228	-6629
351	1695	2016	4811	-791
352	-700	206	-691	999
353	-931	195	-509	254
354	-210	354	24	5106
355	1461	-303	460	1298
356	1767	322	1276	7437
357	-1073	77	3022	-246
358	-328	3909	-5164	13052
359	4962	1555	1346	4890
360	164	-825	148	-664
361	-712	1534	1882	100
362	1639	-1477	1551	719
363	458	1604	1689	681
364	-492	693	2036	-752
365	-2953	3422	12980	1546
366	-1684	1399	-743	2662
367	170	975	2754	3402
368	1001	-1123	965	625
369	-1298	-706	383	-913
370	3957	834	-44	2801
371	578	807	589	314
372	1491	-2300	2439	275
373	383	-1387	863	-617
374	3110	763	-3207	3776
375	350	345	254	272
376	-1470	-415	1447	-2557
377	-709	488	425	-357
378	6649	123	2816	576
379	668	7132	-705	-1755
380	404	1746	9037	-2016
381	78	-579	4192	809
382	1569	4734	2570	621
383	948	2114	-481	471
384	-230	-404	-911	-496
385	364	6114	-3405	772
386	219	-1386	-590	392
387	132	409	-12	327
388	-415	-1352	527	839
389	1262	1249	1092	4365
390	-316	-137	284	-576
391	1052	813	784	682

392	-1133	708	-59	-802
393	-1122	-1	1803	2188
394	-140	-868	-698	-1287
395	30	-191	-227	-322
396	-440	-1237	2506	481
397	-462	222	11632	6914
398	-682	2262	-3326	-7404
399	693	1250	69	1324
400	-837	-670	562	-1320
401	-3340	-763	15	562
402	20	-429	-138	175
403	-54	303	-643	439
404	692	-1589	1010	-97
405	215	51	496	351
406	-404	-569	3822	-821
407	526	2893	1705	5612
408	-1516	262	1503	-3924
409	-1227	-468	1129	836
410	-813	138	-17	-123
411	-209	750	-4825	-3249
412	-789	-180	1749	-1877
413	-359	1774	1935	3252
414	-94	4214	4910	-1833
415	-3882	7957	-5511	3855
416	-367	-47	-342	1252
417	-134	229	-149	-240
418	250	394	-833	716
419	2121	-202	266	2311
420	-4210	116	-748	4067
421	786	-1697	147	2305
422	5624	1248	-5200	3876
423	5066	352	-768	9894
424	204	-382	-154	-238
425	-184	-879	-95	-465
426	-1776	-1346	-649	-43
427	-539	1756	-4159	-445
428	374	-683	-626	-1193
429	-229	1834	-493	2550
430	-1266	149	-2937	1089
431	3233	9153	-9378	3144
432	1149	-1498	174	44
433	-351	332	-22	-165
434	2122	-117	617	378
435	-1944	2443	-3429	403
436	-1496	19	58	2165
437	3673	231	-1501	5085
438	699	3509	2043	2589
439	3559	11798	-3227	5935
440	-658	-1097	-445	-2784
441	-3450	1745	-2433	-2001
442	-6719	3703	-2081	1694
443	-7042	9134	-13671	-2614
444	364	1059	630	-5065
445	-228	6003	-3945	3588
446	847	4540	-6326	742
447	3283	16010	-17227	1636
448	285	-362	322	-672

449	-473	199	-786	-215
450	-369	229	798	-1100
451	182	-2018	-639	-464
452	-561	111	55	-1666
453	583	-2159	1414	751
454	-4419	3446	3790	-7302
455	39	518	-15	-1495
456	-1016	990	3025	-1759
457	-2307	-200	948	-624
458	-2763	2359	1216	-7404
459	-231	752	577	-1353
460	-752	1261	2326	-10115
461	-1315	740	5628	1252
462	-5387	7958	1691	-19224
463	1006	5692	4524	-8230
464	-2944	1850	2262	-1848
465	-1732	-1352	1341	956
466	-194	1366	3307	-500
467	64	-293	192	-355
468	-456	847	1121	-1999
469	-54	-539	525	31
470	2387	9134	9081	-7328
471	-2028	3516	2003	215
472	-9535	708	11389	-7382
473	-7260	3968	5869	228
474	1707	1846	4683	-4601
475	-953	-262	948	96
476	-234	3920	7662	-8159
477	198	731	1058	-1634
478	812	11394	10228	-14268
479	-1258	3393	4573	-4991
480	75	-1149	63	-73
481	-3042	-3091	903	67
482	2452	-2662	1675	1564
483	1612	-3458	53	-543
484	165	-2659	745	2079
485	285	-1048	448	364
486	-690	394	-1441	2126
487	1270	-1203	20	1981
488	-319	-987	689	-240
489	212	-524	354	73
490	-2474	-606	-106	-1153
491	-502	-80	-106	-13
492	-1307	-3030	240	-1958
493	-179	2199	1957	1373
494	-4028	935	-2361	-9619
495	-78	92	-2357	-258
496	2251	-942	2698	1414
497	183	-2076	890	255
498	12708	266	3516	1450
499	2618	-2366	1898	-1164
500	-33	-64	635	197
501	1921	-3074	-837	1172
502	1510	3343	3822	2642
503	328	1840	-568	2628
504	-1948	1732	4074	-774
505	-1101	338	457	252

506	2653	1423	1556	3652
507	-2851	2299	-4021	778
508	-977	702	1439	-1033
509	512	190	-1255	1133
510	3095	2366	4297	-3993
511	5603	6801	-3278	4825
512	3708	-4595	-174	-5466
513	6642	-6292	2437	442
514	5553	-3819	-1853	-8413
515	425	-627	5619	-3556
516	7817	-7075	10600	-2258
517	3925	-2833	-72	-1358
518	11037	-7197	5288	-14434
519	2425	-4350	333	-9257
520	1515	215	-194	90
521	1555	-2325	2488	-2620
522	687	-657	3195	-650
523	3784	-3980	12574	5902
524	1194	-1024	2115	-25
525	-4326	-4919	-7640	326
526	793	-2014	1672	-6939
527	653	-754	2286	318
528	5673	-1116	8246	-2246
529	11688	-8435	16555	2551
530	-1247	-1855	4749	-1838
531	-1005	-7635	17838	-475
532	1301	-1082	2757	-4124
533	9574	-1676	8532	-952
534	2056	-2257	7359	-4384
535	2787	-1039	5899	-11
536	1070	-4111	1342	1460
537	1116	-12475	9995	650
538	1723	-6725	6020	5427
539	-1257	-12863	20202	9518
540	730	2201	-2676	568
541	426	-2249	2616	-2826
542	236	-197	921	2407
543	347	-3829	8218	8722
544	13058	-10488	8453	-9753
545	5056	-2191	4350	-905
546	1778	-15731	4754	-10386
547	2236	-5103	2860	-3991
548	7333	-9303	15598	-16357
549	6027	-13542	2855	-7737
550	61	-12148	12735	-23167
551	-1448	-14926	6518	-13767
552	8335	-2500	-1145	-3713
553	1633	1142	251	2105
554	3704	-2818	4206	-5422
555	-681	-1046	4699	4321
556	5111	-8969	10188	-2045
557	3247	-1963	1582	-2605
558	2955	-7990	14298	-5719
559	1580	-4537	825	-6925
560	4958	-5233	573	-2004
561	4890	-14378	10816	4353
562	2741	-2600	3685	-4183

563	5465	-1216	8887	3905
564	6944	-5380	5414	-7765
565	3224	-4618	1955	-1816
566	-335	-10495	15036	-6311
567	90	-4485	1280	-3781
568	-290	398	-727	2585
569	-1072	-3908	3961	8898
570	1920	-1344	7820	2107
571	-4502	-8827	17295	6552
572	967	-1650	-1808	4294
573	480	-493	750	1257
574	-971	-368	5922	-5544
575	462	-2029	8023	2102
576	782	-470	-269	-853
577	661	-803	223	-123
578	2119	-266	-635	-4444
579	630	1052	315	-942
580	3596	-473	2921	-1915
581	277	356	-759	-478
582	2282	-2421	5053	-10636
583	1276	-818	1150	-2733
584	-355	532	-1715	1003
585	53	1318	-1661	-582
586	-632	2107	217	1528
587	-211	551	2387	1385
588	616	292	-501	-527
589	2509	-364	-5987	-2601
590	-391	-166	236	3245
591	-438	1049	-1580	160
592	5314	-325	-1110	-295
593	4272	-653	7224	-3198
594	1	133	365	-268
595	1713	-1072	5118	100
596	198	287	-48	-211
597	2646	1197	1365	-624
598	673	-1509	1153	-1719
599	236	1532	859	-319
600	-2237	-58	-5065	7544
601	1467	-312	3129	1035
602	-3474	1701	1247	1553
603	-7069	-813	11959	7397
604	61	242	-58	2288
605	588	-131	-1017	-1208
606	-3938	-4608	446	10269
607	-873	-54	2588	3892
608	7524	-2222	2307	-5325
609	912	-628	939	-696
610	-3404	-1697	1102	-10570
611	1136	638	1377	-3107
612	-1671	-2529	3694	-9330
613	697	-3208	-1806	-5118
614	-3035	-3843	15529	-23374
615	3701	-2884	9284	-8114
616	1406	309	1186	-730
617	-62	62	-8	-32
618	-1637	1305	1739	-2757
619	-274	-18	1460	-41

620	2344	-265	1672	-629
621	1109	-3991	-1669	-2016
622	-410	-923	10336	-6935
623	403	-148	1632	-3694
624	1298	807	-252	-1014
625	-325	-3999	3187	3229
626	-1073	-1368	2383	-4904
627	1227	-255	1438	666
628	670	-14	2684	-3137
629	287	-103	-63	-135
630	-255	-3490	11611	-9597
631	-522	-36	2783	-4220
632	-1835	2453	-1409	1043
633	-218	1024	813	2278
634	54	-186	408	884
635	-3969	-4248	5097	4668
636	-26	991	-1669	862
637	19	137	-245	54
638	-1085	-6265	376	-1538
639	108	130	1726	1060
640	1121	-635	236	-988
641	1737	-1161	-771	-1046
642	1419	-1019	760	-1417
643	356	-144	410	-270
644	4517	-438	722	-1664
645	763	-3156	-5529	-9680
646	2411	-1731	-1934	-8028
647	-1668	1040	-1629	-2868
648	6824	-4650	315	2845
649	810	-3890	-5416	-3951
650	1088	249	388	310
651	1714	705	2398	520
652	-1243	-8345	-6597	-302
653	-595	-12918	-13592	-9238
654	15	744	-809	3756
655	-2603	-2817	-8832	-702
656	3589	-2930	-274	1360
657	11450	-7157	1170	474
658	486	-138	202	726
659	198	-3273	5179	-2160
660	649	2464	-410	-1835
661	3013	221	1093	-2541
662	-360	-458	597	1915
663	129	-75	75	-152
664	5599	-10181	1527	13002
665	5539	-2837	1770	1665
666	2093	-1851	4449	5297
667	34	-7568	8611	3506
668	958	-2360	1516	4489
669	-1209	-2368	-7398	-1540
670	-6201	-5468	1745	7751
671	-1834	-204	3621	1843
672	4192	-2003	1263	-9486
673	-48	-101	1465	-1441
674	1987	-5262	-1146	-4680
675	-339	-445	67	-1228
676	9438	-9847	3534	-4707

677	1770	-2859	-992	-3176
678	-6020	-10262	774	-10647
679	12	-1028	1248	-10011
680	2051	420	50	-956
681	-652	684	-1852	-1829
682	93	-186	430	-1151
683	-94	261	2394	-2313
684	2006	-504	1688	-1855
685	-213	-1148	-4752	-7458
686	814	-230	4472	-6898
687	165	-95	3559	-2178
688	1963	2056	-207	-2212
689	2769	-5330	1410	1443
690	-25	-331	-158	-1048
691	827	650	2209	-622
692	5239	-33	1597	-9961
693	972	1168	365	-2043
694	1400	-1650	4458	-4091
695	-1630	-305	3728	-3573
696	5091	-1693	-2809	2600
697	258	485	1732	-264
698	-580	2735	2014	-2558
699	-125	-135	11841	-2969
700	1858	1064	981	-1674
701	2773	-74	4578	-1621
702	-1280	-3555	3417	2632
703	2153	-3308	15960	-2431
704	4241	-958	-416	865
705	3128	2675	610	858
706	-1026	2422	211	571
707	228	3161	-1008	2019
708	20	63	54	41
709	986	1518	-1451	-2295
710	-9	-1388	29	3287
711	456	443	111	-600
712	2355	-38	-8709	6915
713	1567	840	-1919	117
714	-3597	-48	3335	6013
715	-1694	2322	-315	1579
716	180	-1768	1223	4678
717	5114	-3261	-8514	-1950
718	-6386	-179	1249	16705
719	-1122	-813	965	5297
720	10496	-6615	2131	9755
721	7880	-1186	407	-393
722	1599	-20	3765	2851
723	842	123	-476	700
724	900	-493	116	3041
725	1045	407	332	-589
726	747	-5161	3022	10826
727	-4573	-1692	-339	870
728	3881	-3559	-5075	19570
729	2652	-1226	-401	4968
730	-7038	-1565	1483	12077
731	-899	918	616	2329
732	-7936	-3162	-3507	6532
733	-3970	-187	-3887	-1722

734	-12721	-12975	942	19702
735	-12004	-5659	-2488	4690
736	352	-266	911	-1398
737	1089	2159	699	243
738	-2629	586	1150	-2141
739	129	893	-397	-804
740	2755	-289	1942	-3435
741	287	-213	-683	-701
742	-3615	-4600	6882	-6219
743	1395	-23	990	-6559
744	2218	2573	-606	2765
745	677	3951	-1530	-86
746	-130	3203	1032	1364
747	276	3743	919	-457
748	469	-127	-428	45
749	1603	-387	-1020	-80
750	-1750	-1781	1595	3811
751	-157	203	318	-687
752	2340	-127	2231	2730
753	526	-67	-38	-658
754	-2557	1795	-986	113
755	-680	1638	353	266
756	1100	-13	1727	-1545
757	-839	2868	84	-593
758	9	-5786	565	-2024
759	148	-1	83	-655
760	1465	-226	319	5095
761	1974	2118	1354	1007
762	-3150	-2292	-14	2752
763	-780	2824	2501	139
764	-958	-5024	-1323	-252
765	-11	25	-7	-354
766	-9236	-13635	293	4887
767	-1463	-3887	2164	-1698
768	708	-3836	6314	3506
769	1970	-17471	12464	351
770	869	-208	168	-1846
771	1071	-5458	2417	-569
772	1464	-1135	2463	-1733
773	1187	-3095	5791	-1952
774	4159	-3332	734	-3910
775	-2275	-10821	-395	-4807
776	-86	624	1147	246
777	1117	-4502	4199	2764
778	-166	-52	118	-173
779	-449	-3424	3234	-696
780	311	1708	-2741	2083
781	2785	-4153	-3701	-2760
782	1135	-302	-550	1588
783	1184	-971	-9	-4240
784	808	-68	1424	-26
785	5136	-6187	3183	3564
786	7	188	960	-324
787	-748	-1252	7581	-1604
788	14	912	-2205	1789
789	1405	-751	1377	-554
790	648	514	1042	-836

791	-143	-953	515	-1550
792	883	725	-612	3834
793	662	-2368	2651	-1351
794	-1726	-4225	-973	1570
795	-4675	-12839	9323	2710
796	3621	-174	-12424	5116
797	1433	1693	-1521	2069
798	267	-1064	-4460	2164
799	1065	-3217	2198	-1709
800	6483	-74	2695	-5213
801	-2023	-3583	5542	1418
802	-779	-3571	587	-5003
803	-8624	-5214	562	-3138
804	110	-2848	5952	-6395
805	-2292	-3251	2925	-2462
806	-1034	-10647	9901	-8040
807	-14588	-11498	1318	-8601
808	-413	-2206	-934	3276
809	-140	-1706	22	-1385
810	-206	-1621	209	-604
811	-5955	-10295	-72	2099
812	-19	-1128	1935	7769
813	1370	-1144	-2203	-378
814	2795	-669	3004	-3822
815	-7437	-1868	4126	-905
816	400	-26	-554	3070
817	3637	-2283	3713	5056
818	433	-901	-222	712
819	699	-8416	-770	46
820	-1113	-2821	-2692	2759
821	294	-697	-400	-98
822	-1524	-1989	3552	-4000
823	-5160	-4928	-2172	-1541
824	1064	-1709	-2875	15351
825	760	-6388	492	3470
826	2026	-4468	1561	5057
827	-11	-19162	6693	6267
828	-3885	-2603	-9824	16992
829	-985	-1742	-339	6632
830	-2117	-350	-1174	5597
831	-2698	-11997	4125	5799
832	232	-304	1121	-142
833	3770	-4053	6203	-2270
834	2	2321	597	-1057
835	451	-681	671	-599
836	-609	-379	-1198	-1452
837	3582	989	-1566	-1552
838	-531	-1194	1106	-1851
839	-1116	-716	-1651	-2403
840	81	2175	-1785	429
841	2400	-529	-1340	-1498
842	-3614	1796	-546	359
843	-1849	1169	37	-562
844	3540	1102	-2407	391
845	12764	-592	-7511	-2433
846	715	-6994	-3431	7914
847	2516	1617	-2003	1330

848	654	705	-256	-57
849	179	-653	1087	743
850	-3320	318	-676	-253
851	154	473	615	-477
852	735	944	-1342	358
853	503	742	-1803	-1393
854	-310	-1132	-4767	-542
855	-134	4	-288	-12
856	-1170	-828	-3958	1735
857	1877	-636	-180	-724
858	-14629	-473	-2203	5314
859	-4959	-3220	1851	-716
860	3032	-2368	-6663	1038
861	5137	-1085	-2171	-6217
862	-4773	-7174	-12228	6629
863	-1695	-2016	-4811	791
864	700	-206	691	-999
865	931	-195	509	-254
866	210	-354	-24	-5106
867	-1461	303	-460	-1298
868	-1767	-322	-1276	-7437
869	1073	-77	-3022	246
870	328	-3909	5164	-13052
871	-4962	-1555	-1346	-4890
872	-164	825	-148	664
873	712	-1534	-1882	-100
874	-1639	1477	-1551	-719
875	-458	-1604	-1689	-681
876	492	-693	-2036	752
877	2953	-3422	-12980	-1546
878	1684	-1399	743	-2662
879	-170	-975	-2754	-3402
880	-1001	1123	-965	-625
881	1298	706	-383	913
882	-3957	-834	44	-2801
883	-578	-807	-589	-314
884	-1491	2300	-2439	-275
885	-383	1387	-863	617
886	-3110	-763	3207	-3776
887	-350	-345	-254	-272
888	1470	415	-1447	2557
889	709	-488	-425	357
890	-6649	-123	-2816	-576
891	-668	-7132	705	1755
892	-404	-1746	-9037	2016
893	-78	579	-4192	-809
894	-1569	-4734	-2570	-621
895	-948	-2114	481	-471
896	230	404	911	496
897	-364	-6114	3405	-772
898	-219	1386	590	-392
899	-132	-409	12	-327
900	415	1352	-527	-839
901	-1262	-1249	-1092	-4365
902	316	137	-284	576
903	-1052	-813	-784	-682
904	1133	-708	59	802

905	1122	1	-1803	-2188
906	140	868	698	1287
907	-30	191	227	322
908	440	1237	-2506	-481
909	462	-222	-11632	-6914
910	682	-2262	3326	7404
911	-693	-1250	-69	-1324
912	837	670	-562	1320
913	3340	763	-15	-562
914	-20	429	138	-175
915	54	-303	643	-439
916	-692	1589	-1010	97
917	-215	-51	-496	-351
918	404	569	-3822	821
919	-526	-2893	-1705	-5612
920	1516	-262	-1503	3924
921	1227	468	-1129	-836
922	813	-138	17	123
923	209	-750	4825	3249
924	789	180	-1749	1877
925	359	-1774	-1935	-3252
926	94	-4214	-4910	1833
927	3882	-7957	5511	-3855
928	367	47	342	-1252
929	134	-229	149	240
930	-250	-394	833	-716
931	-2121	202	-266	-2311
932	4210	-116	748	-4067
933	-786	1697	-147	-2305
934	-5624	-1248	5200	-3876
935	-5066	-352	768	-9894
936	-204	382	154	238
937	184	879	95	465
938	1776	1346	649	43
939	539	-1756	4159	445
940	-374	683	626	1193
941	229	-1834	493	-2550
942	1266	-149	2937	-1089
943	-3233	-9153	9378	-3144
944	-1149	1498	-174	-44
945	351	-332	22	165
946	-2122	117	-617	-378
947	1944	-2443	3429	-403
948	1496	-19	-58	-2165
949	-3673	-231	1501	-5085
950	-699	-3509	-2043	-2589
951	-3559	-11798	3227	-5935
952	658	1097	445	2784
953	3450	-1745	2433	2001
954	6719	-3703	2081	-1694
955	7042	-9134	13671	2614
956	-364	-1059	-630	5065
957	228	-6003	3945	-3588
958	-847	-4540	6326	-742
959	-3283	-16010	17227	-1636
960	-285	362	-322	672
961	473	-199	786	215

962	369	-229	-798	1100
963	-182	2018	639	464
964	561	-111	-55	1666
965	-583	2159	-1414	-751
966	4419	-3446	-3790	7302
967	-39	-518	15	1495
968	1016	-990	-3025	1759
969	2307	200	-948	624
970	2763	-2359	-1216	7404
971	231	-752	-577	1353
972	752	-1261	-2326	10115
973	1315	-740	-5628	-1252
974	5387	-7958	-1691	19224
975	-1006	-5692	-4524	8230
976	2944	-1850	-2262	1848
977	1732	1352	-1341	-956
978	194	-1366	-3307	500
979	-64	293	-192	355
980	456	-847	-1121	1999
981	54	539	-525	-31
982	-2387	-9134	-9081	7328
983	2028	-3516	-2003	-215
984	9535	-708	-11389	7382
985	7260	-3968	-5869	-228
986	-1707	-1846	-4683	4601
987	953	262	-948	-96
988	234	-3920	-7662	8159
989	-198	-731	-1058	1634
990	-812	-11394	-10228	14268
991	1258	-3393	-4573	4991
992	-75	1149	-63	73
993	3042	3091	-903	-67
994	-2452	2662	-1675	-1564
995	-1612	3458	-53	543
996	-165	2659	-745	-2079
997	-285	1048	-448	-364
998	690	-394	1441	-2126
999	-1270	1203	-20	-1981
1000	319	987	-689	240
1001	-212	524	-354	-73
1002	2474	606	106	1153
1003	502	80	106	13
1004	1307	3030	-240	1958
1005	179	-2199	-1957	-1373
1006	4028	-935	2361	9619
1007	78	-92	2357	258
1008	-2251	942	-2698	-1414
1009	-183	2076	-890	-255
1010	-12708	-266	-3516	-1450
1011	-2618	2366	-1898	1164
1012	33	64	-635	-197
1013	-1921	3074	837	-1172
1014	-1510	-3343	-3822	-2642
1015	-328	-1840	568	-2628
1016	1948	-1732	-4074	774
1017	1101	-338	-457	-252
1018	-2653	-1423	-1556	-3652

1019	2851	-2299	4021	-778
1020	977	-702	-1439	1033
1021	-512	-190	1255	-1133
1022	-3095	-2366	-4297	3993
1023	-5603	-6801	3278	-4825

Table 2.E.7 — Spectral shape (enhabcement layer) cb4k[2][512][4]
dim = 4x512 codewords
16 bits signed
factor = 2^5

index (SE_shape5)	codeword			
0	-412	-994	-739	-344
1	236	-1661	-696	-684
2	-2190	821	-1694	1341
3	-707	1508	61	1107
4	1198	336	93	370
5	-803	459	-843	146
6	1259	3665	436	2673
7	-5678	2246	-1009	947
8	1069	-145	-185	764
9	-52	234	-2248	1307
10	54	723	-828	2248
11	-5190	2683	-3314	4210
12	1910	1707	-1557	4536
13	-3734	4448	-2791	1786
14	681	2683	-3198	10239}
15	-20207	14438	-4021	16989
16	-821	1524	-2028	2000
17	46	263	-961	-1384
18	-5621	3750	-8129	-305
19	-1551	2159	-2029	226
20	-2125	-1176	765	-4
21	261	979	-764	237
22	-1468	1029	-1065	-1167
23	-823	1334	-409	-876
24	-1237	-1215	-1426	-445
25	7	3	117	-388
26	-1877	90	-1748	-2169
27	-451	1766	-647	201
28	1134	643	-735	-105
29	-2354	1324	-886	-2290
30	-604	2078	1214	2999
31	-4053	7276	-2774	306
32	-386	1110	-1488	91
33	670	24	-1432	173
34	-2221	1788	-9716	1702
35	-1343	2122	-3768	2392
36	40	126	-283	-57
37	865	-807	-576	-312
38	1103	2566	-4258	-64
39	-1057	-155	-1320	1680
40	309	31	1	-682
41	36	-1863	-722	170
42	-480	-113	-2137	-861

43	-906	864	-110	2023
44	967	-467	68	375
45	-274	433	-396	1233
46	1733	1607	1388	4135
47	-4658	3707	-2344	4721
48	-2776	4606	-4730	-867
49	1978	1217	-4881	-3162
50	-4620	1838	-13993	-7419
51	-2658	6355	-8561	700
52	-830	863	1077	-2990
53	472	22	-1019	-893
54	-2349	2474	-4673	-6953
55	496	1994	-2199	-2257
56	770	448	-3023	-3298
57	436	-876	-1691	-1622
58	1138	987	-5905	-7911
59	384	1468	-2813	-324
60	216	86	445	-259
61	-476	-148	-158	70
62	1318	178	-2396	-1602
63	-1541	1857	842	1390
64	1104	2194	-991	-1366
65	183	240	-55	-218
66	1676	612	-410	1323
67	2785	515	599	1015
68	-216	96	-849	461
69	281	375	-951	1161
70	167	2080	-4014	3066
71	-1276	-1228	-1647	620
72	-583	-785	-4	-450
73	-485	564	-641	-434
74	139	-150	59	149
75	-1418	-789	-536	1140
76	-320	625	-349	353
77	-1834	1808	-4797	61
78	-634	246	651	3934
79	-6263	1011	-976	3099
80	2044	2788	649	394
81	-15	399	-1583	-494
82	-2441	726	-1402	-99
83	-86	386	225	223
84	-737	1069	-79	-133
85	-2916	3617	-4112	1740
86	975	250	-599	445
87	-2853	1039	-199	550
88	-66	435	-158	-405
89	-1400	3094	-379	-982
90	-523	386	-40	107
91	-2380	4244	1863	-125
92	-809	2683	-2140	-3902
93	-7338	5424	-3747	-5313
94	-3068	375	-431	1535
95	-8049	4056	-672	-3038
96	-1059	9447	-2592	-163
97	-285	3122	415	-518
98	693	3599	-1614	3600
99	-452	108	-1904	627

100	1553	1954	-1475	1034
101	4315	362	-1755	958
102	4709	3342	-8685	5266
103	2318	294	-6746	2268
104	-832	2786	35	327
105	-327	-286	-359	-353
106	206	159	-367	227
107	-827	-1979	-13	-354
108	1294	-427	-997	-283
109	10	137	-742	-12
110	1515	785	-3089	2309
111	-547	1041	-1296	994
112	2724	7267	3470	-3280
113	4307	87	-3989	-2418
114	-454	516	-4247	-2829
115	-976	2313	-1135	1590
116	1965	3341	3746	-1796
117	119	922	-740	-523
118	316	1407	-2417	1308
119	1171	-1377	-1912	-257
120	1645	2575	816	-1268
121	562	-820	-1226	-605
122	1082	142	-1702	-2577
123	-284	342	279	-209
124	-411	-376	-97	-1217
125	-3046	2378	-627	-523
126	83	-66	-87	77
127	-1852	2106	1081	-487
128	232	762	1513	-252
129	-70	-1473	273	-736
130	1608	305	907	323
131	1307	562	446	1448
132	-1495	1065	7	-118
133	-200	-29	480	458
134	142	-122	409	743
135	-1785	3094	-27	3163
136	116	-210	157	-182
137	-418	-35	76	-299
138	47	863	67	218
139	-3150	2681	1193	896
140	99	429	-454	717
141	-2329	1558	1614	2031
142	81	2328	370	4929
143	-9217	4208	3606	8575
144	-56	1395	605	-1845
145	392	-53	-296	-1474
146	-2063	1886	-327	724
147	-95	359	805	97
148	-6063	1753	2742	-3384
149	-749	1189	1409	-1212
150	-679	137	629	-1369
151	-576	5069	3880	-729
152	-208	-1798	717	-1709
153	-1360	-1858	-194	-1201
154	1837	-312	-568	-944
155	-2331	-180	1320	-315
156	-620	511	-138	-2351

157	-646	533	2408	-1218
158	-1855	-1354	1500	160
159	-6498	2601	4574	-776
160	-989	5956	2366	2583
161	-453	1352	544	-776
162	-127	3113	-2159	286
163	-629	-616	-572	77
164	-241	2165	2393	-1884
165	-253	-111	63	-654
166	717	-28	-498	-27
167	-516	228	349	377
168	972	1692	428	2383
169	12	10	-40	-38
170	974	357	-392	-640
171	-1975	-1191	538	1088
172	47	436	304	-584
173	-202	575	229	716
174	107	424	1091	632
175	-4219	2280	3206	3278
176	5301	3994	-60	-2715
177	1568	1015	-309	-6296
178	-3493	1401	-3514	-2795
179	-737	1947	-1687	-1067
180	-5407	4499	3109	-8521
181	-1102	3285	263	-2446
182	-2427	596	964	-1298
183	-345	862	375	-97
184	519	602	792	-885
185	1069	-256	517	-1184
186	1374	1187	-3730	-4497
187	40	-1048	-720	-1240
188	-1030	1415	4368	-2676
189	728	-62	337	-690
190	556	796	-230	-1395
191	-1759	1011	2868	1114
192	1371	5701	528	6
193	3030	1144	869	-29
194	3378	3010	2763	1943
195	4054	1751	3238	3947
196	215	1657	-1001	-282
197	124	47	84	-74
198	979	972	-557	905
199	2015	641	2513	929
200	-4	867	-54	-255
201	34	172	70	71
202	647	281	712	1043
203	1029	55	872	3129
204	1085	-638	-134	-793
205	483	2163	1615	1069
206	921	207	3274	198
207	-939	1034	6613	4879
208	3647	499	4586	-1013
209	3299	1829	-1729	-1373
210	1472	-314	648	-478
211	3224	865	-236	2468
212	140	1736	2784	-3675
213	-109	674	-1718	-1344

214	-463	-907	-1230	-1868
215	-367	494	708	-753
216	316	778	4421	-1091
217	691	15	587	376
218	3163	648	1817	-1144
219	83	308	367	-6
220	1493	2	1757	-2674
221	-1778	1130	90	-1162
222	-300	264	-286	-108
223	-1769	3654	1894	674
224	6094	13540	1477	2528
225	2028	192	1053	-4005
226	1446	8069	2503	4816
227	-323	2370	2421	1159
228	-347	6855	-1048	-3343
229	636	1544	-162	-597
230	3994	2764	-2044	557
231	510	345	-1363	774
232	211	4953	-1633	1087
233	378	559	-550	-81
234	-1555	4061	-93	1843
235	195	353	-69	82
236	-196	1292	-958	-1485
237	69	9	191	255
238	528	471	-25	-389
239	181	824	3944	1653
240	15684	9882	11257	-3356
241	14361	3408	1203	-12054
242	3316	1761	4754	-364
243	4565	344	-1059	-391
244	4585	7075	7707	-13837
245	1228	128	1479	-5643
246	-249	1882	-2060	-6610
247	1507	420	-230	-2383
248	2804	4819	5060	341
249	3754	2544	251	-940
250	1353	855	41	-5
251	972	36	140	-183
252	-3375	1698	1958	-5789
253	-607	522	82	-988
254	-100	551	42	-1312
255	-245	76	168	71
256	412	994	739	344
257	-236	1661	696	684
258	2190	-821	1694	-1341
259	707	-1508	-61	-1107
260	-1198	-336	-93	-370
261	803	-459	843	-146
262	-1259	-3665	-436	-2673
263	5678	-2246	1009	-947
264	-1069	145	185	-764
265	52	-234	2248	-1307
266	-54	-723	828	-2248
267	5190	-2683	3314	-4210
268	-1910	-1707	1557	-4536
269	3734	-4448	2791	-1786
270	-681	-2683	3198	-10239

271	20207	-14438	4021	-16989
272	821	-1524	2028	-2000
273	-46	-263	961	1384
274	5621	-3750	8129	305
275	1551	-2159	2029	-226
276	2125	1176	-765	4
277	-261	-979	764	-237
278	1468	-1029	1065	1167
279	823	-1334	409	876
280	1237	1215	1426	445
281	-7	-3	-117	388
282	1877	-90	1748	2169
283	451	-1766	647	-201
284	-1134	-643	735	105
285	2354	-1324	886	2290
286	604	-2078	-1214	-2999
287	4053	-7276	2774	-306
288	386	-1110	1488	-91
289	-670	-24	1432	-173
290	2221	-1788	9716	-1702
291	1343	-2122	3768	-2392
292	-40	-126	283	57
293	-865	807	576	312
294	-1103	-2566	4258	64
295	1057	155	1320	-1680
296	-309	-31	-1	682
297	-36	1863	722	-170
298	480	113	2137	861
299	906	-864	110	-2023
300	-967	467	-68	-375
301	274	-433	396	-1233
302	-1733	-1607	-1388	-4135
303	4658	-3707	2344	-4721
304	2776	-4606	4730	867
305	-1978	-1217	4881	3162
306	4620	-1838	13993	7419
307	2658	-6355	8561	-700
308	830	-863	-1077	2990
309	-472	-22	1019	893
310	2349	-2474	4673	6953
311	-496	-1994	2199	2257
312	-770	-448	3023	3298
313	-436	876	1691	1622
314	-1138	-987	5905	7911
315	-384	-1468	2813	324
316	-216	-86	-445	259
317	476	148	158	-70
318	-1318	-178	2396	1602
319	1541	-1857	-842	-1390
320	-1104	-2194	991	1366
321	-183	-240	55	218
322	-1676	-612	410	-1323
323	-2785	-515	-599	-1015
324	216	-96	849	-461
325	-281	-375	951	-1161
326	-167	-2080	4014	-3066
327	1276	1228	1647	-620

328	583	785	4	450
329	485	-564	641	434
330	-139	150	-59	-149
331	1418	789	536	-1140
332	320	-625	349	-353
333	1834	-1808	4797	-61
334	634	-246	-651	-3934
335	6263	-1011	976	-3099
336	-2044	-2788	-649	-394
337	15	-399	1583	494
338	2441	-726	1402	99
339	86	-386	-225	-223
340	737	-1069	79	133
341	2916	-3617	4112	-1740
342	-975	-250	599	-445
343	2853	-1039	199	-550
344	66	-435	158	405
345	1400	-3094	379	982
346	523	-386	40	-107
347	2380	-4244	-1863	125
348	809	-2683	2140	3902
349	7338	-5424	3747	5313
350	3068	-375	431	-1535
351	8049	-4056	672	3038
352	1059	-9447	2592	163
353	285	-3122	-415	518
354	-693	-3599	1614	-3600
355	452	-108	1904	-627
356	-1553	-1954	1475	-1034
357	-4315	-362	1755	-958
358	-4709	-3342	8685	-5266
359	-2318	-294	6746	-2268
360	832	-2786	-35	-327
361	327	286	359	353
362	-206	-159	367	-227
363	827	1979	13	354
364	-1294	427	997	283
365	-10	-137	742	12
366	-1515	-785	3089	-2309
367	547	-1041	1296	-994
368	-2724	-7267	-3470	3280
369	-4307	-87	3989	2418
370	454	-516	4247	2829
371	976	-2313	1135	-1590
372	-1965	-3341	-3746	1796
373	-119	-922	740	523
374	-316	-1407	2417	-1308
375	-1171	1377	1912	257
376	-1645	-2575	-816	1268
377	-562	820	1226	605
378	-1082	-142	1702	2577
379	284	-342	-279	209
380	411	376	97	1217
381	3046	-2378	627	523
382	-83	66	87	-77
383	1852	-2106	-1081	487
384	-232	-762	-1513	252

385	70	1473	-273	736
386	-1608	-305	-907	-323
387	-1307	-562	-446	-1448
388	1495	-1065	-7	118
389	200	29	-480	-458
390	-142	122	-409	-743
391	1785	-3094	27	-3163
392	-116	210	-157	182
393	418	35	-76	299
394	-47	-863	-67	-218
395	3150	-2681	-1193	-896
396	-99	-429	454	-717
397	2329	-1558	-1614	-2031
398	-81	-2328	-370	-4929
399	9217	-4208	-3606	-8575
400	56	-1395	-605	1845
401	-392	53	296	1474
402	2063	-1886	327	-724
403	95	-359	-805	-97
404	6063	-1753	-2742	3384
405	749	-1189	-1409	1212
406	679	-137	-629	1369
407	576	-5069	-3880	729
408	208	1798	-717	1709
409	1360	1858	194	1201
410	-1837	312	568	944
411	2331	180	-1320	315
412	620	-511	138	2351
413	646	-533	-2408	1218
414	1855	1354	-1500	-160
415	6498	-2601	-4574	776
416	989	-5956	-2366	-2583
417	453	-1352	-544	776
418	127	-3113	2159	-286
419	629	616	572	-77
420	241	-2165	-2393	1884
421	253	111	-63	654
422	-717	28	498	27
423	516	-228	-349	-377
424	-972	-1692	-428	-2383
425	-12	-10	40	38
426	-974	-357	392	640
427	1975	1191	-538	-1088
428	-47	-436	-304	584
429	202	-575	-229	-716
430	-107	-424	-1091	-632
431	4219	-2280	-3206	-3278
432	-5301	-3994	60	2715
433	-1568	-1015	309	6296
434	3493	-1401	3514	2795
435	737	-1947	1687	1067
436	5407	-4499	-3109	8521
437	1102	-3285	-263	2446
438	2427	-596	-964	1298
439	345	-862	-375	97
440	-519	-602	-792	885
441	-1069	256	-517	1184

442	-1374	-1187	3730	4497
443	-40	1048	720	1240
444	1030	-1415	-4368	2676
445	-728	62	-337	690
446	-556	-796	230	1395
447	1759	-1011	-2868	-1114
448	-1371	-5701	-528	-6
449	-3030	-1144	-869	29
450	-3378	-3010	-2763	-1943
451	-4054	-1751	-3238	-3947
452	-215	-1657	1001	282
453	-124	-47	-84	74
454	-979	-972	557	-905
455	-2015	-641	-2513	-929
456	4	-867	54	255
457	-34	-172	-70	-71
458	-647	-281	-712	-1043
459	-1029	-55	-872	-3129
460	-1085	638	134	793
461	-483	-2163	-1615	-1069
462	-921	-207	-3274	-198
463	939	-1034	-6613	-4879
464	-3647	-499	-4586	1013
465	-3299	-1829	1729	1373
466	-1472	314	-648	478
467	-3224	-865	236	-2468
468	-140	-1736	-2784	3675
469	109	-674	1718	1344
470	463	907	1230	1868
471	367	-494	-708	753
472	-316	-778	-4421	1091
473	-691	-15	-587	-376
474	-3163	-648	-1817	1144
475	-83	-308	-367	6
476	-1493	-2	-1757	2674
477	1778	-1130	-90	1162
478	300	-264	286	108
479	1769	-3654	-1894	-674
480	-6094	-13540	-1477	-2528
481	-2028	-192	-1053	4005
482	-1446	-8069	-2503	-4816
483	323	-2370	-2421	-1159
484	347	-6855	1048	3343
485	-636	-1544	162	597
486	-3994	-2764	2044	-557
487	-510	-345	1363	-774
488	-211	-4953	1633	-1087
489	-378	-559	550	81
490	1555	-4061	93	-1843
491	-195	-353	69	-82
492	196	-1292	958	1485
493	-69	-9	-191	-255
494	-528	-471	25	389
495	-181	-824	-3944	-1653
496	-15684	-9882	-11257	3356
497	-14361	-3408	-1203	12054
498	-3316	-1761	-4754	364

499	-4565	-344	1059	391
500	-4585	-7075	-7707	13837
501	-1228	-128	-1479	5643
502	249	-1882	2060	6610
503	-1507	-420	230	2383
504	-2804	-4819	-5060	-341
505	-3754	-2544	-251	940
506	-1353	-855	-41	5
507	-972	-36	-140	183
508	3375	-1698	-1958	5789
509	607	-522	-82	988
510	100	-551	-42	1312
511	245	-76	-168	-71

Table 2.E.8 — Spectral shape (enhabcement layer): cb4k[3][64][4]
dim = 4x64 codewords
16 bits signed
factor = 2^6

index (SE_shape6)	codeword			
0	-3768	947	-5129	3159
1	1856	645	-1585	744
2	-4337	2591	1792	1066
3	-4965	-846	-951	-897
4	-8691	3396	-5050	-2288
5	-2722	-2820	-2120	387
6	-1439	-2113	-366	-1372
7	-3042	265	1742	-5266
8	777	338	487	285
9	8269	4345	-3265	-4617
10	-1473	51	9119	7231
11	-132	126	126	136
12	-899	-707	-2724	-2446
13	1298	2196	-4573	-505
14	-1113	869	733	5361
15	654	-787	-1161	-806
16	-1026	9637	-5916	4157
17	2689	405	86	-1898
18	-6583	15055	4584	5480
19	-5119	3610	8620	-5558
20	-1484	1303	-485	-738
21	390	1243	-86	-628
22	-2005	3752	-2611	-3901
23	283	6610	6930	-15838
24	2401	5663	5252	3885
25	23296	5077	7584	-6529
26	-1723	3993	283	1313
27	558	-1877	2107	-2168
28	-1	379	-887	599
29	1464	5776	-374	-3465
30	38	-1486	-3389	1175
31	326	736	-98	-2403
32	3768	-947	5129	-3159
33	-1856	-645	1585	-744
34	4337	-2591	-1792	-1066

35	4965	846	951	897
36	8691	-3396	5050	2288
37	2722	2820	2120	-387
38	1439	2113	366	1372
39	3042	-265	-1742	5266
40	-777	-338	-487	-285
41	-8269	-4345	3265	4617
42	1473	-51	-9119	-7231
43	132	-126	-126	-136
44	899	707	2724	2446
45	-1298	-2196	4573	505
46	1113	-869	-733	-5361
47	-654	787	1161	806
48	1026	-9637	5916	-4157
49	-2689	-405	-86	1898
50	6583	-15055	-4584	-5480
51	5119	-3610	-8620	5558
52	1484	-1303	485	738
53	-390	-1243	86	628
54	2005	-3752	2611	3901
55	-283	-6610	-6930	15838
56	-2401	-5663	-5252	-3885
57	-23296	-5077	-7584	6529
58	1723	-3993	-283	-1313
59	-558	1877	-2107	2168
60	1	-379	887	-599
61	-1464	-5776	374	3465
62	-38	1486	3389	-1175
63	-326	-736	98	2403

2.E.4 CbCelp

VQ codebook for stochastic excitation vector for 2 kbps

Table 2.E.9 — 1st stage VXC gain codebook (base layer): cbL0_g[16]
dim = 1 x 16 codewords
16 bits signed
factor = 2^2

index (VX_gain1[])	codeword
0	65
1	164
2	325
3	764
4	1505
5	2069
6	2428
7	3257
8	4132
9	4687
10	5729
11	6294
12	7137
13	8037
14	11402
15	18768

Table 2.E.10 — 1st stage VXC shape codebook(base layer): cbL0_s[64][80]
dim = 80 x 64 codewords
16 bits signed
factor = 2^15

index (VX_ shape1[])	codeword									
0	0	0	0	1628	0	-9648	0	0	0	0
	0	0	0	0	0	1446	0	0	0	-886
	0	-5869	5431	0	0	0	0	0	-5434	423
	0	0	-10553	0	0	-1118	0	0	0	0
	0	0	0	0	-2550	0	0	0	0	0
	26	0	350	0	0	-339	5478	-541	0	0
	0	0	0	0	2070	-16335	9139	0	2613	-15972
	0	0	9563	0	3610	0	-116	-3282	0	0
1	0	0	0	1475	2107	0	755	0	0	-3955
	-2615	-965	-407	-708	1825	0	-5847	3465	420	-6981
	0	0	-2308	2285	-2949	0	0	6434	-7241	-3798
	3568	1179	0	4944	0	0	-4981	3712	-2667	-6447
	5312	-9873	0	3222	-6426	0	669	5199	0	0
	412	-2163	2657	0	0	1597	2207	2172	-7297	0
	5462	-971	-3483	749	0	2047	-3053	0	0	-11458
	-2981	0	-4517	3871	-931	4693	10075	2245	-2730	6256
2	0	-3703	4128	676	0	-874	4680	-4808	1978	-8956
	1205	8163	0	0	-466	-3607	7256	0	0	-617
	0	0	2287	0	1506	-1742	1919	0	986	5364
	-3468	0	493	4803	0	1028	550	2746	0	0
	116	6542	0	480	1829	-3433	999	-10662	-8148	-985
	-3087	4157	-309	3235	-10231	5358	-4104	-3156	0	4481
	-2203	0	5019	0	6958	0	0	-5297	0	-557
	-4389	0	5956	623	-3635	-3938	344	-2245	5708	1950
3	0	0	1073	-2434	-6808	2035	0	-3928	0	-2511
	-2238	3122	4694	-12171	0	0	-1316	0	552	0
	-1752	2655	693	-948	5612	3864	-196	1977	0	3197
	-2269	776	-2206	605	-901	-1997	-1395	502	-9523	2649
	3943	-2173	-647	325	-3185	-1515	199	-11174	-1384	-6662
	-1582	0	0	-3042	4265	4346	7434	5001	-789	1679
	1369	655	-6038	2194	-1661	0	5080	3424	-1932	98
	-2938	11836	6427	0	0	918	0	-1182	0	-1693
4	-4632	0	336	-2778	-4905	-3720	57	195	6479	4647
	6028	-3904	3466	0	3156	-1882	1249	-2516	-106	-770
	2434	-7902	7161	-1952	-8165	0	-7560	0	0	-5449
	0	2652	0	696	0	0	-2030	-1140	4057	-2308
	1796	-4008	0	0	0	-1746	0	1361	0	1440
	-8255	2610	-5339	-463	-2478	-4294	-3816	-4121	0	-6542
	0	-4338	-576	0	0	7433	0	5051	-1554	-8066
	7895	-2736	7067	289	2992	0	0	4541	11	0
5	0	0	0	0	0	0	-2440	0	2741	0
	0	0	4142	0	0	0	5335	0	10364	0
	0	0	0	-5110	0	0	0	13977	-90	-2187
	2840	0	10564	0	0	0	0	0	0	0
	0	4350	-3172	1911	2832	0	0	-4587	0	2496
	-4029	-309	3503	6785	0	676	235	0	0	0
	0	0	11368	0	-8166	0	0	0	0	-1013

	-5017	0	143	0	286	-10526	0	0	9233	0
	2801	3312	0	3537	-2895	3251	-2901	-1217	0	-3317
	-2481	-2216	-626	1046	723	-514	0	-1160	0	7014
	0	-5144	-5483	2965	4999	-3799	-6616	-4771	-4315	-3333
6	7797	-488	0	3845	1670	2468	5530	7787	0	-1978
	6397	-2176	0	522	-3652	652	-1728	2135	-3578	0
	-71	0	-11335	1286	-8933	-2911	-377	847	0	-1155
	3491	-1048	-1559	5575	4336	8164	0	0	0	-581
	0	-764	0	-372	-3988	-3748	0	9410	0	-2331
	-5896	2103	0	0	0	0	0	0	0	0
	0	0	-1125	0	1197	0	0	1459	0	7113
	-4932	0	3371	1837	0	0	0	259	0	-17178
7	0	1971	11477	0	-367	-9599	0	3890	3020	0
	0	-3923	0	0	0	0	0	-2635	0	0
	0	0	0	0	5266	-2257	0	2639	0	6862
	0	-2905	846	0	0	7696	-6590	-293	0	0
	0	8117	0	9712	3390	0	0	0	0	0
	-1065	-5252	-2075	3686	3042	-2580	556	-5459	4931	-1895
	-3616	162	-392	5018	-1610	-101	2791	8132	38	-4170
	6590	5939	0	4939	0	2589	3788	-197	0	5150
8	2915	-4902	0	-1952	-6332	-6494	-4614	5070	0	6238
	-1369	213	-3010	-579	745	2480	0	1121	-8112	0
	0	-7134	-4975	0	0	5318	1336	0	-516	-5017
	0	0	8462	0	0	564	0	4592	-9288	0
	-1438	1780	1410	0	-1208	-3160	1072	2857	5309	-3738
	849	0	1435	-552	0	-8434	1680	1106	3764	-1977
	98	0	-5953	0	14849	4047	-3176	0	130	0
	3853	-3289	-5691	5670	1809	-1264	-329	0	-1706	0
9	-852	-11144	3495	-3522	-2924	4615	-2256	960	-705	-2819
	266	-3377	-2016	0	0	0	1344	2383	7518	-971
	1613	1710	-3127	-2451	-2077	2992	-10137	0	-265	2079
	0	-8393	0	2089	446	-334	0	-2613	0	-153
	3342	0	4900	-1937	-7294	1668	0	1973	0	-235
	0	-3054	1210	55	0	10118	3822	-1164	4682	1485
	-3077	1877	-5951	0	-3389	-3625	2156	0	-3513	0
	-4489	5203	4396	0	-1647	0	-443	-1805	0	4814
10	7914	-696	0	-973	0	0	-2873	3341	2900	0
	4343	-5076	1170	638	-2612	-1071	0	0	-2392	0
	3912	-7665	2586	-322	-2691	-202	-10380	-9307	4649	3806
	-1057	1274	1739	0	-592	7202	-1524	5719	0	-789
	5937	7425	5298	-2991	0	1659	-4841	283	3365	-2322
	0	0	-1134	0	3516	0	-4115	4584	1049	-731
	0	0	392	-5397	3288	0	4564	-11448	0	1214
	0	-2453	-867	0	2145	-1018	0	1303	0	1205
11	-1938	-1729	371	0	0	1984	265	469	-4587	-3683
	-4090	0	-786	5759	-3493	7663	-685	2011	-1585	0
	5208	4688	2324	-2484	-983	0	-141	-9515	-2796	4548
	3400	-4490	2241	-173	321	5670	-176	-482	10008	-1300
	3679	5685	6846	6285	-7339	-60	4848	-1152	1883	-8428
	5050	-2352	0	-2587	-33	-5587	-7823	3089	273	1542
	-1374	0	638	2094	2421	-1455	6703	4091	4090	-6905
	3113	0	-1907	-1358	0	-558	0	0	176	12935
12	-2019	-4243	0	0	185	-2409	-412	0	-2999	541
	0	-4530	1876	0	0	-8879	-862	-1789	9535	-347
	-4275	3231	0	189	-3061	-3445	-1998	-782	1297	0
	3723	-109	2703	-1393	-3715	3781	-614	-6610	0	779
	3172	-6262	-1869	1864	-3723	9765	0	2667	3084	6209

	0	0	-6233	0	4287	5387	2358	1427	-3202	4462
	-1146	-91	436	3305	-684	4828	3859	3954	10108	-167
	-7522	824	-6239	-1820	0	2383	3595	141	-5560	697
13	-5941	-3208	-3586	-2679	3457	2050	0	-6246	-5370	-892
	0	6372	-2173	4247	0	-3670	4879	1486	447	-107
	-3250	2538	1010	-4783	-1306	-2882	-1413	0	2587	1033
	6844	-46	4332	0	0	-1890	2144	6195	-2670	4559
	-1876	0	2828	-9121	0	-2064	0	-8753	-412	-1143
	0	0	0	9266	0	-4289	0	0	15630	0
	0	0	0	0	0	0	0	0	-3536	0
	0	0	0	0	0	1043	0	0	-1938	-3441
14	0	0	0	0	1315	3492	6433	12535	-15578	0
	3885	0	0	6219	-5423	3477	0	0	0	0
	0	0	0	-2157	0	0	-5362	0	0	0
	1369	0	4980	0	0	0	-986	0	0	0
	6508	0	0	0	0	0	-6175	2022	0	0
	0	1000	0	0	0	4115	-411	0	0	0
	0	0	0	0	4732	0	0	0	0	0
	-552	-1705	0	0	0	2493	4927	-1114	1317	0
15	7065	0	0	0	4801	0	0	-5461	0	0
	0	0	0	21047	0	0	0	0	0	12979
	0	0	0	1009	0	0	-4331	0	0	4513
	0	0	5996	0	0	-763	0	5993	-1291	0
	-7022	8196	0	5302	0	-4124	-2552	0	0	0
	0	0	0	0	-5383	-7389	0	0	0	0
	0	-13760	0	0	-3669	0	0	-1643	0	0
	0	2977	0	4044	0	0	0	0	0	6383
16	13648	0	-6472	0	13024	4114	0	0	-1042	0
	0	0	0	0	0	0	3187	5351	0	0
	0	0	8936	0	0	0	0	0	4985	0
	1943	-8121	4195	0	0	-7224	0	-839	0	0
	0	0	0	241	-1024	4255	0	0	0	0
	0	-352	-9216	-2509	0	1151	0	1168	0	6570
	-830	0	0	0	-2908	8076	0	5816	5142	0
	7360	-713	-7775	-276	42	-885	0	6374	3988	0
17	-1463	3099	0	1129	-296	1663	-9442	-1374	395	-2783
	0	0	0	0	-6757	-432	-2752	268	2439	27
	814	0	8202	1398	3575	5765	0	5918	-4320	5177
	-343	-3570	0	0	-362	0	-3256	9844	-3848	0
	-5975	781	-6308	5389	-564	-2132	0	1289	0	3434
	11485	4693	-1464	0	1169	-373	-2657	-1288	-3197	0
	2976	819	-5121	0	290	-6066	3512	0	-4592	-3678
	-5017	7883	2549	-2245	0	0	-2861	0	0	3968
18	0	-105	8171	0	-182	-5712	-1732	-2874	-51	-7855
	5330	-4002	0	2438	2085	128	-7971	647	1517	-1079
	-2742	8536	4959	-11	351	2243	314	3201	0	-4812
	4864	-3055	-693	-271	0	0	0	6266	-709	4732
	-6127	740	-3295	553	1130	0	4146	-6180	2141	3239
	0	0	0	0	0	0	0	0	-6369	0
	0	0	-8103	-12722	0	0	0	0	0	-5372
	0	0	4509	0	8314	0	0	0	6881	0
19	-2386	0	0	0	0	0	-985	0	0	0
	0	0	-1170	0	0	0	0	0	0	0
	14144	0	0	0	3394	5389	11441	0	0	0
	0	0	0	0	0	0	0	6490	0	0
	-11175	-7384	-944	-51	0	5487	-2815	0	0	0
	0	0	7331	-30	772	-3345	-4	5323	0	0

	-3825	861	4892	-3041	0	0	-5795	4004	2822	-3616
	-3113	-1839	0	-3353	-2123	4453	2619	-2365	0	-573
20	0	8823	4250	4137	11473	5642	0	4533	-1725	0
	0	1444	789	1955	0	-939	2634	5827	-2601	0
	3843	3417	-4439	8355	0	-1822	13	-1928	2517	6680
	-1708	0	0	0	2300	-1011	0	-3620	6077	-9616
	0	-2230	0	2865	1787	0	-2575	-475	-9613	1683
	-6370	3044	-1614	5027	0	0	-188	-4008	4652	1786
	0	0	-3383	-80	-397	773	5044	-2376	-3987	2986
	-2588	0	-2188	7185	-2573	38	7660	6425	4679	0
21	-4949	-7600	0	0	4828	-4138	-735	115	3320	-2284
	-455	4452	0	0	-678	2518	-2983	1463	9195	2537
	1221	3618	5296	5192	1556	981	-2289	0	-372	0
	-203	-508	343	5721	-9816	-8157	0	511	6950	-1274
	2501	-217	-1725	-8232	-953	0	-4249	1549	-1561	-1298
	-4120	-2452	0	0	0	-7557	-253	5271	-4395	0
	1383	1664	-448	0	-6496	0	8286	6876	-2217	-2241
	-5546	-4159	3094	89	1928	-1582	-2991	-6757	0	8154
22	-6394	1902	-3271	2294	5468	8427	110	1715	-445	8136
	0	0	0	7386	-1552	8494	0	0	2300	-689
	-3332	-3630	0	-138	1099	-3992	-913	946	0	0
	0	-3311	0	0	-5438	3781	-4147	0	-1997	-7183
	2832	-1887	-3664	251	147	0	2943	1761	-295	-3906
	-5107	0	0	5554	0	-7204	0	0	-3108	10958
	2828	-2865	-4753	-3109	-5593	-3208	-1682	-1097	-4323	0
	0	0	4268	0	-2139	1085	0	-4253	3418	2898
23	0	2676	-567	-4796	178	0	1409	0	6034	36
	0	-354	6967	-2796	8976	7085	0	3037	2837	0
	4467	2711	-5297	-1146	-6279	-2830	5912	-4656	1005	-2739
	0	-320	1205	-6012	-5827	277	2814	-838	3587	1270
	-959	-5659	-851	-2368	3944	0	-6922	0	4475	0
	2041	-2004	5807	-7	0	93	0	-804	0	2145
	-3096	-443	1422	4705	0	-1012	-1070	-3094	-3329	3558
	-5845	2769	6684	13751	-159	-4829	0	0	4992	-630
24	-394	-1387	870	-5289	0	1702	6045	-2847	638	0
	120	2919	2205	5628	-2922	0	3884	1216	-2743	-8517
	0	0	0	-2379	3830	3287	-5517	0	-3794	1250
	0	-5108	0	6727	3890	0	8902	2978	0	-2671
	378	2132	-7835	2072	-7198	3121	3499	0	-701	0
	-1084	-9811	-2812	307	-1685	1395	2176	874	938	2237
	0	0	2626	-1287	-2104	-1192	5465	1341	-279	0
	9960	4426	-3028	-6230	-8698	0	-480	-2983	2306	1797
25	215	-4147	-323	455	-4713	0	-1598	-12536	0	-3033
	-1293	5039	-4209	-2357	3772	3194	-1317	-6097	-97	0
	3311	0	2859	-3491	-946	-2342	-2145	2822	2197	-785
	-1301	-6387	-4577	0	6077	-2620	-2301	2787	-5771	359
	3002	-2689	-2601	5782	-1777	1149	0	-902	-2232	-6885
	-1706	3623	-7171	5205	-717	819	0	-5399	-3431	-9984
	0	-895	-812	2398	0	-230	-6871	0	5805	0
	0	-1062	6539	2553	-3109	-511	-8288	1209	4089	3174
26	1641	0	-1375	6902	1479	4840	129	-3638	0	-5979
	0	0	2057	1653	3548	-4513	1652	0	-5528	0
	0	2222	-318	1322	4640	8416	-1588	0	5503	-6732
	0	0	2958	-3042	0	3508	9357	2262	4697	4754
	602	455	2598	0	1652	5161	-1436	0	1153	-2628
	0	0	-4122	2261	2255	5431	165	232	0	-1523
	2961	8009	1679	0	2634	5995	-5033	2076	4643	3632

	10186	-2828	3728	0	-3280	-264	-2782	2747	-4782	4130
27	3962	1547	-2122	0	3027	0	0	-5551	1812	-3402
	-1021	2948	5856	-3754	5926	0	-832	-4974	1819	3626
	4830	758	0	0	0	7293	0	-7381	-1682	2552
	2267	-3943	-3057	-77	0	-7395	0	0	0	8258
	-1749	5276	911	-1313	6356	101	7532	1735	0	0
	9072	-1778	3454	0	7230	4572	0	-5294	2407	4007
	-460	0	-3073	7235	-2091	0	0	0	530	2369
	345	-906	-5179	-464	-4748	8088	-5287	-2219	5722	-1440
28	-3436	-303	765	445	3825	0	0	1555	-6168	7294
	-4325	-161	-2464	0	-3557	4776	6953	1438	0	2214
	3138	8154	3604	226	0	-4286	-98	-1317	0	0
	6270	883	-1838	4342	1714	0	543	-7746	0	-5677
	0	0	605	1272	-6247	4822	0	-406	-5400	0
	-1097	0	-3924	-8023	0	2634	0	0	0	0
	-6244	0	-10975	-2833	0	0	0	-5461	0	2082
	0	0	0	-3415	-9918	1542	4593	0	0	0
29	0	5352	0	0	-8238	0	0	-12856	0	0
	0	3890	-4495	0	0	0	0	-2047	0	0
	0	0	-5080	0	0	-4531	0	0	0	-12455
	1336	2662	-3712	-473	0	-3681	4013	0	0	-2858
	0	30	0	0	-1423	0	0	0	-9231	0
	-1128	0	3372	2295	-3313	-7195	1280	0	1432	418
	0	-2962	-3011	1226	5502	4299	9680	6982	66	6894
	0	-1637	3362	5338	1142	467	1384	-2865	0	0
30	1018	1847	368	11959	0	7847	4692	-4576	-3965	-2424
	-1024	4240	-1882	-1208	0	-3357	0	-2708	0	-1789
	-5718	-2094	1961	791	0	0	0	0	-3306	1323
	2281	940	-6635	4014	0	4353	2789	0	4408	3033
	7680	3426	6043	1490	3924	4250	-2864	0	-2128	5098
	0	0	-3227	0	-3958	67	9489	-2570	-2475	-2381
	-3205	-4690	-8091	0	-2284	-58	0	0	0	-1318
	5077	2795	4787	0	-3518	-707	-515	1896	-10246	563
31	6495	3516	5934	4480	976	12457	0	-2461	0	-1943
	-473	5375	0	3354	620	3095	-2340	-1644	1973	0
	0	3	-593	0	-689	0	5169	-2305	-3561	4592
	-1724	2567	-2833	0	7678	3137	636	0	-896	-3601
	-1221	-866	6375	3067	-341	945	-5307	-4781	6805	-2
	0	0	0	-1628	0	9648	0	0	0	0
	0	0	0	0	0	-1446	0	0	0	886
	0	5869	-5431	0	0	0	0	0	5434	-423
32	0	0	10553	0	0	1118	0	0	0	0
	0	0	0	0	2550	0	0	0	0	0
	-26	0	-350	0	0	339	-5478	541	0	0
	0	0	0	0	-2070	16335	-9139	0	-2613	15972
	0	0	-9563	0	-3610	0	116	3282	0	0
	0	0	0	-1475	-2107	0	-755	0	0	3955
	2615	965	407	708	-1825	0	5847	-3465	-420	6981
	0	0	2308	-2285	2949	0	0	-6434	7241	3798
33	-3568	-1179	0	-4944	0	0	4981	-3712	2667	6447
	-5312	9873	0	-3222	6426	0	-669	-5199	0	0
	-412	2163	-2657	0	0	-1597	-2207	-2172	7297	0
	-5462	971	3483	-749	0	-2047	3053	0	0	11458
	2981	0	4517	-3871	931	-4693	-10075	-2245	2730	-6256
	0	3703	-4128	-676	0	874	-4680	4808	-1978	8956
	-1205	-8163	0	0	466	3607	-7256	0	0	617
	0	0	-2287	0	-1506	1742	-1919	0	-986	-5364

34	3468	0	-493	-4803	0	-1028	-550	-2746	0	0
	-116	-6542	0	-480	-1829	3433	-999	10662	8148	985
	3087	-4157	309	-3235	10231	-5358	4104	3156	0	-4481
	2203	0	-5019	0	-6958	0	0	5297	0	557
	4389	0	-5956	-623	3635	3938	-344	2245	-5708	-1950
35	0	0	-1073	2434	6808	-2035	0	3928	0	2511
	2238	-3122	-4694	12171	0	0	1316	0	-552	0
	1752	-2655	-693	948	-5612	-3864	196	-1977	0	-3197
	2269	-776	2206	-605	901	1997	1395	-502	9523	-2649
	-3943	2173	647	-325	3185	1515	-199	11174	1384	6662
	1582	0	0	3042	-4265	-4346	-7434	-5001	789	-1679
	-1369	-655	6038	-2194	1661	0	-5080	-3424	1932	-98
	2938	-11836	-6427	0	0	-918	0	1182	0	1693
36	4632	0	-336	2778	4905	3720	-57	-195	-6479	-4647
	-6028	3904	-3466	0	-3156	1882	-1249	2516	106	770
	-2434	7902	-7161	1952	8165	0	7560	0	0	5449
	0	-2652	0	-696	0	0	2030	1140	-4057	2308
	-1796	4008	0	0	0	1746	0	-1361	0	-1440
	8255	-2610	5339	463	2478	4294	3816	4121	0	6542
	0	4338	576	0	0	-7433	0	-5051	1554	8066
	-7895	2736	-7067	-289	-2992	0	0	-4541	-11	0
37	0	0	0	0	0	0	2440	0	-2741	0
	0	0	-4142	0	0	0	-5335	0	-10364	0
	0	0	0	5110	0	0	0	-13977	90	2187
	-2840	0	-10564	0	0	0	0	0	0	0
	0	-4350	3172	-1911	-2832	0	0	4587	0	-2496
	4029	309	-3503	-6785	0	-676	-235	0	0	0
	0	0	-11368	0	8166	0	0	0	0	1013
	5017	0	-143	0	-286	10526	0	0	-9233	0
38	-2801	-3312	0	-3537	2895	-3251	2901	1217	0	3317
	2481	2216	626	-1046	-723	514	0	1160	0	-7014
	0	5144	5483	-2965	-4999	3799	6616	4771	4315	3333
	-7797	488	0	-3845	-1670	-2468	-5530	-7787	0	1978
	-6397	2176	0	-522	3652	-652	1728	-2135	3578	0
	71	0	11335	-1286	8933	2911	377	-847	0	1155
	-3491	1048	1559	-5575	-4336	-8164	0	0	0	581
	0	764	0	372	3988	3748	0	-9410	0	2331
39	5896	-2103	0	0	0	0	0	0	0	0
	0	0	1125	0	-1197	0	0	-1459	0	-7113
	4932	0	-3371	-1837	0	0	0	-259	0	17178
	0	-1971	-11477	0	367	9599	0	-3890	-3020	0
	0	3923	0	0	0	0	0	2635	0	0
	0	0	0	0	-5266	2257	0	-2639	0	-6862
	0	2905	-846	0	0	-7696	6590	293	0	0
	0	-8117	0	-9712	-3390	0	0	0	0	0
40	1065	5252	2075	-3686	-3042	2580	-556	5459	-4931	1895
	3616	-162	392	-5018	1610	101	-2791	-8132	-38	4170
	-6590	-5939	0	-4939	0	-2589	-3788	197	0	-5150
	-2915	4902	0	1952	6332	6494	4614	-5070	0	-6238
	1369	-213	3010	579	-745	-2480	0	-1121	8112	0
	0	7134	4975	0	0	-5318	-1336	0	516	5017
	0	0	-8462	0	0	-564	0	-4592	9288	0
	1438	-1780	-1410	0	1208	3160	-1072	-2857	-5309	3738
41	-849	0	-1435	552	0	8434	-1680	-1106	-3764	1977
	-98	0	5953	0	-14849	-4047	3176	0	-130	0
	-3853	3289	5691	-5670	-1809	1264	329	0	1706	0
	852	11144	-3495	3522	2924	-4615	2256	-960	705	2819

	-266	3377	2016	0	0	0	-1344	-2383	-7518	971
	-1613	-1710	3127	2451	2077	-2992	10137	0	265	-2079
	0	8393	0	-2089	-446	334	0	2613	0	153
	-3342	0	-4900	1937	7294	-1668	0	-1973	0	235
	0	3054	-1210	-55	0	-10118	-3822	1164	-4682	-1485
	3077	-1877	5951	0	3389	3625	-2156	0	3513	0
	4489	-5203	-4396	0	1647	0	443	1805	0	-4814
42	-7914	696	0	973	0	0	2873	-3341	-2900	0
	-4343	5076	-1170	-638	2612	1071	0	0	2392	0
	-3912	7665	-2586	322	2691	202	10380	9307	-4649	-3806
	1057	-1274	-1739	0	592	-7202	1524	-5719	0	789
	-5937	-7425	-5298	2991	0	-1659	4841	-283	-3365	2322
	0	0	1134	0	-3516	0	4115	-4584	-1049	731
	0	0	-392	5397	-3288	0	-4564	11448	0	-1214
	0	2453	867	0	-2145	1018	0	-1303	0	-1205
43	1938	1729	-371	0	0	-1984	-265	-469	4587	3683
	4090	0	786	-5759	3493	-7663	685	-2011	1585	0
	-5208	-4688	-2324	2484	983	0	141	9515	2796	-4548
	-3400	4490	-2241	173	-321	-5670	176	482	-10008	1300
	-3679	-5685	-6846	-6285	7339	60	-4848	1152	-1883	8428
	-5050	2352	0	2587	33	5587	7823	-3089	-273	-1542
	1374	0	-638	-2094	-2421	1455	-6703	-4091	-4090	6905
	-3113	0	1907	1358	0	558	0	0	-176	-12935
44	2019	4243	0	0	-185	2409	412	0	2999	-541
	0	4530	-1876	0	0	8879	862	1789	-9535	347
	4275	-3231	0	-189	3061	3445	1998	782	-1297	0
	-3723	109	-2703	1393	3715	-3781	614	6610	0	-779
	-3172	6262	1869	-1864	3723	-9765	0	-2667	-3084	-6209
	0	0	6233	0	-4287	-5387	-2358	-1427	3202	-4462
	1146	91	-436	-3305	684	-4828	-3859	-3954	-10108	167
	7522	-824	6239	1820	0	-2383	-3595	-141	5560	-697
45	5941	3208	3586	2679	-3457	-2050	0	6246	5370	892
	0	-6372	2173	-4247	0	3670	-4879	-1486	-447	107
	3250	-2538	-1010	4783	1306	2882	1413	0	-2587	-1033
	-6844	46	-4332	0	0	1890	-2144	-6195	2670	-4559
	1876	0	-2828	9121	0	2064	0	8753	412	1143
	0	0	0	-9266	0	4289	0	0	-15630	0
	0	0	0	0	0	0	0	0	3536	0
	0	0	0	0	0	-1043	0	0	1938	3441
46	0	0	0	0	-1315	-3492	-6433	-12535	15578	0
	-3885	0	0	-6219	5423	-3477	0	0	0	0
	0	0	0	2157	0	0	5362	0	0	0
	-1369	0	-4980	0	0	0	986	0	0	0
	-6508	0	0	0	0	0	6175	-2022	0	0
	0	-1000	0	0	0	-4115	411	0	0	0
	0	0	0	0	-4732	0	0	0	0	0
	552	1705	0	0	0	-2493	-4927	1114	-1317	0
47	-7065	0	0	0	-4801	0	0	5461	0	0
	0	0	0	-21047	0	0	0	0	0	-12979
	0	0	0	-1009	0	0	4331	0	0	-4513
	0	0	-5996	0	0	763	0	-5993	1291	0
	7022	-8196	0	-5302	0	4124	2552	0	0	0
	0	0	0	0	5383	7389	0	0	0	0
	0	13760	0	0	3669	0	0	1643	0	0
	0	-2977	0	-4044	0	0	0	0	0	-6383
48	-13648	0	6472	0	-13024	-4114	0	0	1042	0
	0	0	0	0	0	0	-3187	-5351	0	0

	0	0	-8936	0	0	0	0	0	-4985	0
	-1943	8121	-4195	0	0	7224	0	839	0	0
	0	0	0	-241	1024	-4255	0	0	0	0
	0	352	9216	2509	0	-1151	0	-1168	0	-6570
	830	0	0	0	2908	-8076	0	-5816	-5142	0
	-7360	713	7775	276	-42	885	0	-6374	-3988	0
49	1463	-3099	0	-1129	296	-1663	9442	1374	-395	2783
	0	0	0	0	6757	432	2752	-268	-2439	-27
	-814	0	-8202	-1398	-3575	-5765	0	-5918	4320	-5177
	343	3570	0	0	362	0	3256	-9844	3848	0
	5975	-781	6308	-5389	564	2132	0	-1289	0	-3434
	-11485	-4693	1464	0	-1169	373	2657	1288	3197	0
	-2976	-819	5121	0	-290	6066	-3512	0	4592	3678
	5017	-7883	-2549	2245	0	0	2861	0	0	-3968
50	0	105	-8171	0	182	5712	1732	2874	51	7855
	-5330	4002	0	-2438	-2085	-128	7971	-647	-1517	1079
	2742	-8536	-4959	11	-351	-2243	-314	-3201	0	4812
	-4864	3055	693	271	0	0	0	-6266	709	-4732
	6127	-740	3295	-553	-1130	0	-4146	6180	-2141	-3239
	0	0	0	0	0	0	0	0	6369	0
	0	0	8103	12722	0	0	0	0	0	5372
	0	0	-4509	0	-8314	0	0	0	-6881	0
51	2386	0	0	0	0	0	985	0	0	0
	0	0	1170	0	0	0	0	0	0	0
	-14144	0	0	0	-3394	-5389	-11441	0	0	0
	0	0	0	0	0	0	0	-6490	0	0
	11175	7384	944	51	0	-5487	2815	0	0	0
	0	0	-7331	30	-772	3345	4	-5323	0	0
	3825	-861	-4892	3041	0	0	5795	-4004	-2822	3616
	3113	1839	0	3353	2123	-4453	-2619	2365	0	573
52	0	-8823	-4250	-4137	-11473	-5642	0	-4533	1725	0
	0	-1444	-789	-1955	0	939	-2634	-5827	2601	0
	-3843	-3417	4439	-8355	0	1822	-13	1928	-2517	-6680
	1708	0	0	0	-2300	1011	0	3620	-6077	9616
	0	2230	0	-2865	-1787	0	2575	475	9613	-1683
	6370	-3044	1614	-5027	0	0	188	4008	-4652	-1786
	0	0	3383	80	397	-773	-5044	2376	3987	-2986
	2588	0	2188	-7185	2573	-38	-7660	-6425	-4679	0
53	4949	7600	0	0	-4828	4138	735	-115	-3320	2284
	455	-4452	0	0	678	-2518	2983	-1463	-9195	-2537
	-1221	-3618	-5296	-5192	-1556	-981	2289	0	372	0
	203	508	-343	-5721	9816	8157	0	-511	-6950	1274
	-2501	217	1725	8232	953	0	4249	-1549	1561	1298
	4120	2452	0	0	0	7557	253	-5271	4395	0
	-1383	-1664	448	0	6496	0	-8286	-6876	2217	2241
	5546	4159	-3094	-89	-1928	1582	2991	6757	0	-8154
54	6394	-1902	3271	-2294	-5468	-8427	-110	-1715	445	-8136
	0	0	0	-7386	1552	-8494	0	0	-2300	689
	3332	3630	0	138	-1099	3992	913	-946	0	0
	0	3311	0	0	5438	-3781	4147	0	1997	7183
	-2832	1887	3664	-251	-147	0	-2943	-1761	295	3906
	5107	0	0	-5554	0	7204	0	0	3108	-10958
	-2828	2865	4753	3109	5593	3208	1682	1097	4323	0
	0	0	-4268	0	2139	-1085	0	4253	-3418	-2898
55	0	-2676	567	4796	-178	0	-1409	0	-6034	-36
	0	354	-6967	2796	-8976	-7085	0	-3037	-2837	0
	-4467	-2711	5297	1146	6279	2830	-5912	4656	-1005	2739

	0	320	-1205	6012	5827	-277	-2814	838	-3587	-1270
	959	5659	851	2368	-3944	0	6922	0	-4475	0
	-2041	2004	-5807	7	0	-93	0	804	0	-2145
	3096	443	-1422	-4705	0	1012	1070	3094	3329	-3558
	5845	-2769	-6684	-13751	159	4829	0	0	-4992	630
56	394	1387	-870	5289	0	-1702	-6045	2847	-638	0
	-120	-2919	-2205	-5628	2922	0	-3884	-1216	2743	8517
	0	0	0	2379	-3830	-3287	5517	0	3794	-1250
	0	5108	0	-6727	-3890	0	-8902	-2978	0	2671
	-378	-2132	7835	-2072	7198	-3121	-3499	0	701	0
	1084	9811	2812	-307	1685	-1395	-2176	-874	-938	-2237
	0	0	-2626	1287	2104	1192	-5465	-1341	279	0
	-9960	-4426	3028	6230	8698	0	480	2983	-2306	-1797
57	-215	4147	323	-455	4713	0	1598	12536	0	3033
	1293	-5039	4209	2357	-3772	-3194	1317	6097	97	0
	-3311	0	-2859	3491	946	2342	2145	-2822	-2197	785
	1301	6387	4577	0	-6077	2620	2301	-2787	5771	-359
	-3002	2689	2601	-5782	1777	-1149	0	902	2232	6885
	1706	-3623	7171	-5205	717	-819	0	5399	3431	9984
	0	895	812	-2398	0	230	6871	0	-5805	0
	0	1062	-6539	-2553	3109	511	8288	-1209	-4089	-3174
58	-1641	0	1375	-6902	-1479	-4840	-129	3638	0	5979
	0	0	-2057	-1653	-3548	4513	-1652	0	5528	0
	0	-2222	318	-1322	-4640	-8416	1588	0	-5503	6732
	0	0	-2958	3042	0	-3508	-9357	-2262	-4697	-4754
	-602	-455	-2598	0	-1652	-5161	1436	0	-1153	2628
	0	0	4122	-2261	-2255	-5431	-165	-232	0	1523
	-2961	-8009	-1679	0	-2634	-5995	5033	-2076	-4643	-3632
	-10186	2828	-3728	0	3280	264	2782	-2747	4782	-4130
59	-3962	-1547	2122	0	-3027	0	0	5551	-1812	3402
	1021	-2948	-5856	3754	-5926	0	832	4974	-1819	-3626
	-4830	-758	0	0	0	-7293	0	7381	1682	-2552
	-2267	3943	3057	77	0	7395	0	0	0	-8258
	1749	-5276	-911	1313	-6356	-101	-7532	-1735	0	0
	-9072	1778	-3454	0	-7230	-4572	0	5294	-2407	-4007
	460	0	3073	-7235	2091	0	0	0	-530	-2369
	-345	906	5179	464	4748	-8088	5287	2219	-5722	1440
60	3436	303	-765	-445	-3825	0	0	-1555	6168	-7294
	4325	161	2464	0	3557	-4776	-6953	-1438	0	-2214
	-3138	-8154	-3604	-226	0	4286	98	1317	0	0
	-6270	-883	1838	-4342	-1714	0	-543	7746	0	5677
	0	0	-605	-1272	6247	-4822	0	406	5400	0
	1097	0	3924	8023	0	-2634	0	0	0	0
	6244	0	10975	2833	0	0	0	5461	0	-2082
	0	0	0	3415	9918	-1542	-4593	0	0	0
61	0	-5352	0	0	8238	0	0	12856	0	0
	0	-3890	4495	0	0	0	0	2047	0	0
	0	0	5080	0	0	4531	0	0	0	12455
	-1336	-2662	3712	473	0	3681	-4013	0	0	2858
	0	-30	0	0	1423	0	0	0	9231	0
	1128	0	-3372	-2295	3313	7195	-1280	0	-1432	-418
	0	2962	3011	-1226	-5502	-4299	-9680	-6982	-66	-6894
	0	1637	-3362	-5338	-1142	-467	-1384	2865	0	0
62	-1018	-1847	-368	-11959	0	-7847	-4692	4576	3965	2424
	1024	-4240	1882	1208	0	3357	0	2708	0	1789
	5718	2094	-1961	-791	0	0	0	0	3306	-1323
	-2281	-940	6635	-4014	0	-4353	-2789	0	-4408	-3033

	-7680	-3426	-6043	-1490	-3924	-4250	2864	0	2128	-5098
	0	0	3227	0	3958	-67	-9489	2570	2475	2381
	3205	4690	8091	0	2284	58	0	0	0	1318
	-5077	-2795	-4787	0	3518	707	515	-1896	10246	-563
63	-6495	-3516	-5934	-4480	-976	-12457	0	2461	0	1943
	473	-5375	0	-3354	-620	-3095	2340	1644	-1973	0
	0	-3	593	0	689	0	-5169	2305	3561	-4592
	1724	-2567	2833	0	-7678	-3137	-636	0	896	3601
	1221	866	-6375	-3067	341	-945	5307	4781	-6805	2

2.E.5 CbCelp4k

VQ codebook for stochastic excitation vector for 4 kbps

Table 2.E.11 — 2nd stage VXC gain (enhancement layer) codebook cbL1_g[8]
dim = 1 x 8 codewords
16 bits signed
factor = 2^1

index (VX_gain2[])	codeword
0	29
1	351
2	1523
3	820
4	16956
5	2571
6	7670
7	4303

Table 2.E.12 — 2nd stage VXC shape (enhancement layer) codebook cbL1_s[32][40]
dim = 40 x 32 codewords
16 bits signed
factor = 2^15

index (VX_ shape2[])	codeword									
0	0	0	0	0	-3670	0	-3592	6877	0	0
	0	0	0	-1882	0	0	0	5892	0	5063
	0	0	0	0	0	11204	0	0	0	0
	-619	-6147	0	0	0	0	9657	8329	16851	-18053
1	0	0	0	-3118	0	0	-5467	0	9244	0
	2584	0	1675	0	5550	-786	5419	2660	0	15964
	-14118	-15419	0	0	0	0	-1030	0	0	0
	2789	1168	5227	0	0	0	0	-7551	0	-9213
2	1105	2945	7478	0	2906	542	-538	-1123	6571	-9293
	8215	5999	5572	0	1720	-2450	12025	3924	-8051	-2883
	2211	0	3123	-299	5563	2605	8868	2438	2353	6959
	-242	-675	-1992	-5682	-4249	12192	-4202	0	6805	-5750
3	54	9003	0	2103	0	266	-7156	-846	3992	0
	4324	5100	-6345	0	1619	-3191	0	6368	0	0
	0	-6450	-1637	-2510	-4786	-8345	-4638	-7092	-4275	4220

	16349	-4636	0	0	-10748	0	8287	0	0	-7619
	0	0	0	0	0	0	0	0	0	0
4	0	-30522	0	-6518	0	0	0	0	0	0
	0	0	0	2300	0	3129	0	0	0	0
	0	0	0	5230	0	0	0	7566	0	0
	0	-532	0	6930	0	0	0	0	0	59
5	0	-8218	0	0	1674	0	-24232	0	0	712
	0	0	0	0	0	0	0	0	0	0
	0	10185	-3778	9	-3862	-7654	0	0	-13262	0
	0	0	0	0	-2098	0	0	-19805	0	0
6	795	-10197	0	0	0	0	0	10192	-11014	0
	-1663	0	0	0	2713	0	0	0	0	0
	0	0	0	0	-18106	0	3056	0	0	0
	0	-1230	0	0	4856	15371	0	-7639	0	16108
7	7534	0	0	0	0	0	4774	0	3514	0
	0	0	-1939	0	0	-4330	0	0	-16240	0
	0	0	0	0	-8721	-6353	0	0	0	0
	-7583	-949	0	1701	-4425	0	0	-8468	-7271	0
8	1000	3659	-1614	5090	-6310	-5811	0	-535	13517	0
	0	3450	-1640	0	2814	-4112	-2567	6184	508	10094
	0	-1806	5543	-1061	-4553	0	-12870	-3778	10647	5770
	-5533	-9268	0	5815	-456	0	0	3105	5292	8540
9	2957	11480	-877	1426	-1279	4972	3974	5746	-614	15998
	-8941	404	-6202	0	0	3109	-4787	-3417	1754	0
	3235	-6437	7917	4443	-3728	725	-4565	0	-1519	0
	0	0	0	0	0	0	0	15292	0	4112
10	-3166	-131	20232	0	0	0	-7397	0	0	0
	0	15002	0	7723	0	0	0	5995	0	0
	0	3968	0	-3532	0	0	0	0	0	0
	0	1009	0	-6702	-793	7142	419	7381	6528	1006
11	-10093	10645	-11016	735	0	-7898	-3148	426	494	2241
	13739	7757	3838	-753	-4526	0	6304	0	0	-5553
	-1579	-2655	0	3192	0	0	-4345	-2671	-7876	0
	0	-2889	-207	240	4040	5354	0	-12614	0	-4645
12	3752	-14607	4174	-2397	1174	0	1403	3077	6548	0
	10126	0	-2811	0	2615	8341	6045	0	2943	0
	-341	0	10856	4636	5544	-11308	346	189	1374	1064
	0	-17688	14233	7304	3965	1493	0	0	0	0
13	-3907	-2200	0	0	1184	0	0	0	0	0
	0	-4015	0	0	0	0	0	0	0	0
	0	17747	0	0	-4870	0	10521	0	0	0
	-1201	-7227	-8405	0	2758	4653	-3605	-4825	-4317	-7995
14	938	-7482	190	4838	3207	0	-2449	-8396	1109	-10123
	-225	13603	5322	5193	6497	-920	4448	587	-122	-1543
	6005	-4309	1921	3173	7250	5770	0	-8388	0	0
	0	-3983	0	-2562	19732	0	0	0	0	0
15	0	0	-6400	0	0	-11621	2715	0	0	-1657
	0	6695	0	-2777	0	-11764	8241	0	-13969	0
	2647	0	0	0	0	2337	0	0	3062	0
	0	0	0	0	3670	0	3592	-6877	0	0
16	0	0	0	1882	0	0	0	-5892	0	-5063
	0	0	0	0	0	-11204	0	0	0	0
	619	6147	0	0	0	0	-9657	-8329	-16851	18053
	0	0	0	3118	0	0	5467	0	-9244	0
17	-2584	0	-1675	0	-5550	786	-5419	-2660	0	-15964
	14118	15419	0	0	0	0	1030	0	0	0
	-2789	-1168	-5227	0	0	0	0	7551	0	9213

	-1105	-2945	-7478	0	-2906	-542	538	1123	-6571	9293
18	-8215	-5999	-5572	0	-1720	2450	-12025	-3924	8051	2883
	-2211	0	-3123	299	-5563	-2605	-8868	-2438	-2353	-6959
	242	675	1992	5682	4249	-12192	4202	0	-6805	5750
	-54	-9003	0	-2103	0	-266	7156	846	-3992	0
19	-4324	-5100	6345	0	-1619	3191	0	-6368	0	0
	0	6450	1637	2510	4786	8345	4638	7092	4275	-4220
	-16349	4636	0	0	10748	0	-8287	0	0	7619
	0	0	0	0	0	0	0	0	0	0
20	0	30522	0	6518	0	0	0	0	0	0
	0	0	0	-2300	0	-3129	0	0	0	0
	0	0	0	-5230	0	0	0	-7566	0	0
	0	532	0	-6930	0	0	0	0	0	-59
21	0	8218	0	0	-1674	0	24232	0	0	-712
	0	0	0	0	0	0	0	0	0	0
	0	-10185	3778	-9	3862	7654	0	0	13262	0
	0	0	0	0	2098	0	0	19805	0	0
22	-795	10197	0	0	0	0	0	-10192	11014	0
	1663	0	0	0	-2713	0	0	0	0	0
	0	0	0	0	18106	0	-3056	0	0	0
	0	1230	0	0	-4856	-15371	0	7639	0	-16108
23	-7534	0	0	0	0	0	-4774	0	-3514	0
	0	0	1939	0	0	4330	0	0	16240	0
	0	0	0	0	8721	6353	0	0	0	0
	7583	949	0	-1701	4425	0	0	8468	7271	0
24	-1000	-3659	1614	-5090	6310	5811	0	535	-13517	0
	0	-3450	1640	0	-2814	4112	2567	-6184	-508	-10094
	0	1806	-5543	1061	4553	0	12870	3778	-10647	-5770
	5533	9268	0	-5815	456	0	0	-3105	-5292	-8540
25	-2957	-11480	877	-1426	1279	-4972	-3974	-5746	614	-15998
	8941	-404	6202	0	0	-3109	4787	3417	-1754	0
	-3235	6437	-7917	-4443	3728	-725	4565	0	1519	0
	0	0	0	0	0	0	0	-15292	0	-4112
26	3166	131	-20232	0	0	0	7397	0	0	0
	0	-15002	0	-7723	0	0	0	-5995	0	0
	0	-3968	0	3532	0	0	0	0	0	0
	0	-1009	0	6702	793	-7142	-419	-7381	-6528	-1006
27	10093	-10645	11016	-735	0	7898	3148	-426	-494	-2241
	-13739	-7757	-3838	753	4526	0	-6304	0	0	5553
	1579	2655	0	-3192	0	0	4345	2671	7876	0
	0	2889	207	-240	-4040	-5354	0	12614	0	4645
28	-3752	14607	-4174	2397	-1174	0	-1403	-3077	-6548	0
	-10126	0	2811	0	-2615	-8341	-6045	0	-2943	0
	341	0	-10856	-4636	-5544	11308	-346	-189	-1374	-1064
	0	17688	-14233	-7304	-3965	-1493	0	0	0	0
29	3907	2200	0	0	-1184	0	0	0	0	0
	0	4015	0	0	0	0	0	0	0	0
	0	-17747	0	0	4870	0	-10521	0	0	0
	1201	7227	8405	0	-2758	-4653	3605	4825	4317	7995
30	-938	7482	-190	-4838	-3207	0	2449	8396	-1109	10123
	225	-13603	-5322	-5193	-6497	920	-4448	-587	122	1543
	-6005	4309	-1921	-3173	-7250	-5770	0	8388	0	0
	0	3983	0	2562	-19732	0	0	0	0	0
31	0	0	6400	0	0	11621	-2715	0	0	1657
	0	-6695	0	2777	0	11764	-8241	0	13969	0
	-2647	0	0	0	0	-2337	0	0	-3062	0

2.E.6 CbLsp

Table 2.E.13 — LSP table for the first stage (base layer) lsp_tbl[0][32][10]
dim = 10 x 32 codevectors
16 bits signed
factor = 2^15

index (LSP1)	codeword									
0	3375	4912	6549	9040	10942	13276	18798	21307	25336	28572
1	2974	4358	6129	8373	10084	12442	19149	23195	26224	28839
2	3077	4360	6147	9751	12072	14239	16729	19410	25495	28508
3	2919	4067	6571	9049	11301	13176	15546	21792	25596	28758
4	3935	6019	9204	12175	15469	18353	21731	24243	27476	29137
5	3746	5894	8346	10956	14067	16427	19980	22820	26182	28388
6	3138	4830	6689	9712	13093	14804	17683	20287	23614	27858
7	3660	5585	8408	11300	13434	15086	18394	21960	25381	27908
8	2546	3478	5301	11498	13855	15746	18993	21490	26264	28407
9	2415	3239	5561	12877	15707	17387	20352	22286	26337	28318
10	2887	5150	8479	11388	14513	17282	20858	23623	26958	29134
11	2229	4162	7949	11474	14612	16990	19987	22614	25791	28349
12	2086	3932	8403	12430	16065	19196	22252	24785	27464	29264
13	3539	5949	9973	13626	17153	20423	23144	25356	28016	29594
14	3748	6180	9973	12764	15130	17300	20631	23046	26361	28654
15	4005	6856	12057	14541	17344	19943	22286	24277	26824	28540
16	2959	4902	6772	9100	12970	14348	17631	22669	24817	27497
17	1659	3244	6690	9923	13244	16000	19579	22730	26119	28767
18	2116	2880	4959	10683	16288	18759	21936	24273	27098	29082
19	1623	2982	7079	10655	14022	17143	20470	23340	26641	29021
20	2954	4057	5645	9387	15193	16955	19540	22376	24759	27631
21	1829	2579	4149	9097	13989	16374	19707	22537	26415	29208
22	2395	3424	5074	8394	15219	18639	20682	22949	25314	27660
23	2155	2832	4495	7776	13914	18654	22078	24004	26849	28673
24	2853	4870	7173	10343	12233	14846	19653	22280	26468	28773
25	2952	4252	6548	8892	10909	16697	19529	21846	25191	27573
26	2438	3437	5374	7557	9440	15048	19085	22361	26340	29023
27	2745	3740	5868	8032	10298	16927	21146	23387	26806	28435
28	2212	3253	5397	9031	13431	15972	19009	20587	24238	28524
29	2327	3469	5224	7762	13556	15354	18303	22448	25218	28360
30	2114	3126	4926	9253	11770	14176	19094	21896	25847	28980
31	1877	2610	4300	7363	12346	16129	19777	22688	26019	29255

Table 2.E.14 — LSP table for the second stage of VQ (base layer) with inter-frame prediction
pd_tbl[0][64][5]....Lower 5 LSPs
dim = 5 x 64 codevectors
16 bits signed
factor = 2^18

index (LSP2&0x3f)	codeword				
0	2981	1974	3673	2079	3154
1	467	1177	2624	4315	-2970
2	7204	-2351	-225	2548	1793
3	3151	1499	1654	7626	8950
4	1476	2753	-2614	5096	4202

5	2121	4158	5054	-4650	-4200
6	3688	5151	590	7476	-4205
7	-435	1111	1001	-257	3523
8	199	1958	-2865	-1864	4551
9	-2645	-3880	1767	6223	2994
10	2283	5489	4357	412	-7054
11	-1025	668	-181	4580	713
12	506	-401	-3597	8910	9823
13	-1358	-388	624	3479	5877
14	-501	-886	-506	10302	4008
15	-4779	3673	611	3201	3169
16	1979	3589	779	1408	7565
17	2129	6236	2383	5214	1688
18	5956	11047	-2496	1672	-1735
19	1969	1523	1908	7647	1473
20	5550	9285	7505	6436	5741
21	6357	13996	6349	-2076	-2341
22	7812	4527	1326	6386	2207
23	3594	4936	-2939	1523	288
24	755	1649	3576	-1211	-401
25	-239	-1389	-2181	765	7891
26	1617	852	3819	15383	92
27	792	-1261	2456	3584	1098
28	1575	1148	3979	7198	-8126
29	2071	1214	-574	1085	307
30	3397	501	4493	1132	-3308
31	-844	794	6753	8625	-1594
32	440	-1832	3555	-1777	2459
33	797	-1148	-936	1547	3481
34	3329	477	-2144	5308	-2666
35	149	-4244	8061	3492	739
36	855	2485	7891	-6944	404
37	4160	6139	-4802	-5122	3500
38	637	1712	6758	7673	4768
39	-63	-2758	1751	8480	-2330
40	-2645	-3741	2750	-225	7649
41	-2021	-4108	6286	11770	7086
42	-8	3906	1311	1583	-1916
43	-3064	-2095	-1437	4685	10533
44	-1667	6748	8748	2755	-1358
45	-60	994	5023	2697	9461
46	-2593	-121	4202	2273	771
47	-4401	-724	12926	3948	3434
48	1274	4462	894	-2286	1214
49	4538	9141	-1080	962	6478
50	5405	1950	1001	-4055	551
51	1395	4590	16809	6637	1025
52	4538	6252	6019	-252	2855
53	8968	5665	1282	-199	-2789
54	3869	3712	6493	4913	-933
55	4045	2713	438	304	-3185
56	-1114	3025	6289	-367	3639
57	907	-1691	-3196	5450	2254
58	1311	2721	-3743	551	13002
59	-1077	10402	2066	-398	907
60	1211	1067	1772	-5211	5101
61	3953	66	-3337	-1368	5204

62	744	629	1077	-2087	11838
63	-304	-299	8247	-414	-3557

Table 2.E.15 — LSP table for the second stage of VQ (base layer) with inter-frame prediction
pd_tbl[1][16][5]....Higher 5 LSPs
dim = 5 x 16 codevectors
16 bits signed
factor = 2^18

index (LSP3&0x0f))	codeword				
0	9524	346	-2981	267	-336
1	2629	-5468	-1733	7104	2469
2	2336	3321	-2915	6391	2404
3	-3961	-370	1623	9820	4352
4	9849	6079	3437	7157	2553
5	3586	755	1937	-1298	2530
6	5240	8829	-624	-1243	-464
7	-2223	3743	-31	1688	5419
8	6027	-2265	2632	3738	-1646
9	-792	-3861	5951	2705	-160
10	671	2637	1455	1216	-2724
11	-4336	2679	7699	1557	-186
12	9484	2750	3806	-2873	-4066
13	3054	4257	8559	6189	1531
14	1922	4716	7917	-3945	-2375
15	-881	10551	3992	3806	2097

Table 2.E.16 — LSP table for the second stage of VQ (base layer) without inter-frame prediction
d_tbl[0][64][5] ...Lower 5 LSPs
dim = 5 x 64 codevectors
16 bits signed
factor = 2^18

index (LSP2&0x3f))	codeword				
0	-223	1541	5665	3589	4611
1	-1733	5350	5348	-2451	2902
2	4365	1696	-1562	10659	-2362
3	2954	3694	8486	1274	1578
4	4567	7319	2747	1699	8294
5	-47	2110	650	2165	983
6	7083	9013	-2564	-8268	3416
7	3979	278	4509	2923	-1523
8	-2687	-4168	10231	5379	2732
9	-1269	-2312	15862	4470	-5888
10	1927	383	9885	10635	102
11	3358	5584	194	4881	-9241
12	-29	-81	-454	3785	7969
13	3743	6973	7581	-6653	637
14	3510	3856	8617	4687	-6399
15	10040	9558	5277	2257	13
16	-3358	2847	9891	3261	-1167
17	-6430	-1893	9864	11762	6879

18	-191	1827	2063	7943	-3277
19	-1903	-3143	5510	-446	7222
20	-16	-2212	-3951	10250	4797
21	-4687	548	2074	7198	3906
22	2074	1735	-1153	10121	12945
23	-377	-419	6189	10315	-8486
24	-7073	1169	3332	2551	9020
25	-4939	-8029	17592	12255	703
26	-2564	6800	2024	4650	71
27	-1458	6008	13668	5980	5924
28	-3209	-5445	-3395	14132	7644
29	-687	7188	8622	-2095	-6593
30	-912	2008	-5607	6719	3109
31	3513	6559	6921	8247	1853
32	4669	17485	10142	-1971	-2771
33	3135	5544	-954	-9437	8593
34	10533	126	-6483	-3484	419
35	8284	3707	-1059	3534	-2933
36	10785	12945	-1140	-8242	-8908
37	5337	-89	-3880	-404	8646
38	6491	6700	-1067	-2202	1523
39	2485	-1591	3112	-2773	4467
40	1725	2511	7398	1536	11975
41	3169	5062	886	6984	3715
42	6868	4021	3720	-3122	-6226
43	8420	8389	-8045	734	322
44	5730	2192	1583	2847	5028
45	6637	-4359	1064	6286	2658
46	1316	2881	3332	-2060	-3492
47	3445	6693	2477	328	-1851
48	448	-3172	4742	3854	760
49	-1769	-1250	-6430	2102	13361
50	4126	666	-2931	336	477
51	3101	3133	1699	-4653	3374
52	5838	996	-8074	6229	3678
53	328	294	8632	-553	-4315
54	844	-147	4902	10163	7196
55	-2157	821	-881	-2902	7178
56	-2763	-5652	1966	6113	8562
57	-1442	-4297	1481	8989	385
58	1667	2842	-637	-3439	13508
59	3287	4981	-8630	2231	10179
60	338	2173	10124	-7749	3300
61	1201	5500	-3513	747	3932
62	-3038	12727	2823	-4664	2621
63	-1376	-1489	634	15435	-1937

Table 2.E.17 — LSP table for the second stage of VQ (base layer) without inter-frame prediction
d_tbl[1][16][5] ...Higher 5 LSPs
dim = 5 x 16 codevectors
16 bits signed
factor = 2^18

index (LSP3&0x0f)	codeword				
0	9875	-2490	3812	946	-2427

1	3125	-3028	7267	8889	954
2	11600	5883	2380	-3463	-4367
3	2878	8431	9744	-3673	-4653
4	3400	1762	-202	-380	-1056
5	-6367	2517	7846	2265	-58
6	8800	10082	7710	7075	2535
7	2302	6040	3728	6978	1756
8	2412	-3458	-2760	10473	4608
9	-7044	-2173	3982	11235	7959
10	11136	1680	-4422	5780	2074
11	1387	965	4569	802	2606
12	-1717	5264	-3919	6189	5738
13	-2530	8092	12281	9626	4338
14	5219	9477	-2813	-852	1308
15	-4401	12027	3458	-1256	1859

2.E.7 CbLsp4k

Table 2.E.18 — VQ codebook for LSP quantization of enhancement layer for 4kbps vqLsp[256][10].
dim = 10 x 256 codevectors
16 bits signed
factor = 2^18

index (LSP5)	codeword									
0	-1488	1955	6590	-1047	-422	-567	-311	46	-916	-397
1	445	1443	1785	6945	2534	806	2409	1184	707	4339
2	553	2225	250	390	3316	4497	-67	-2833	-169	3250
3	1219	1234	1126	639	1238	1503	5514	4088	258	4326
4	-716	-923	1856	-2189	1003	-851	-1693	2519	-3541	-1953
5	3885	-4033	-1171	1778	1840	-201	423	-254	-1660	110
6	532	-674	26	-1549	531	896	-1359	-2189	-817	2616
7	1160	-652	-444	879	-264	1034	1506	-637	-759	5774
8	-284	-767	2894	163	463	532	-1775	-1743	1438	3535
9	14	1248	1582	1472	7825	-41	-1292	-738	638	2979
10	1723	-1233	-729	-2381	2160	951	-366	2293	2087	2766
11	633	-1517	-1386	-104	4742	1236	-553	2077	-1963	-19
12	-889	-1170	1810	-2026	1300	1412	2235	-2597	-365	-73
13	-315	-1085	1493	-804	4255	581	-384	-1215	4135	1961
14	-367	-1149	843	-3832	-751	1773	-141	1380	2936	1253
15	-1436	-2531	-1351	2589	3187	548	-322	1020	2293	235
16	-3201	-3223	7383	297	-543	651	-678	-323	1849	103
17	279	-953	-88	4524	1394	985	-368	-162	3737	869
18	140	1243	1575	1246	1088	12797	-2644	1357	1423	3826
19	276	1147	-496	2601	816	5761	-409	2353	180	3808
20	-1839	850	1755	987	-1721	-136	-1827	1149	1978	2236
21	-1593	-989	2006	1295	-178	383	370	2375	2045	-1645
22	676	706	442	-406	582	3248	-2396	-1511	102	-834
23	958	743	-1318	380	143	1849	618	2813	646	-691
24	-1454	-1321	534	-226	-169	498	-728	-3274	6877	4855
25	-2038	907	756	503	2000	-682	-511	-1216	455	2455
26	654	-1055	80	-932	1317	7334	-623	-987	2224	1816
27	-1838	934	-1394	1145	1012	1086	30	-1165	3178	5932
28	-650	-1000	439	28	855	1335	309	3629	8164	22246

29	-74	624	981	-277	2096	949	-252	1934	2831	1794
30	-390	-824	10	-502	180	1024	-3594	1312	453	13048
31	-441	-689	-748	339	1238	1602	-256	1273	4056	1139
32	-488	847	1551	-1291	-1350	-2360	1121	1975	2557	1389
33	-28	-1340	-598	4263	267	-583	339	2073	-5695	-1087
34	604	-1471	-121	-810	-204	556	8401	9515	-1066	529
35	-990	1082	-143	2570	2955	846	3306	2630	-2567	-953
36	612	-870	816	-2482	-2483	-141	219	-1016	-8297	-4247
37	1846	-970	-555	1981	70	36	-506	-3256	-4191	-3182
38	277	-1461	1531	-1314	-77	-841	5243	-549	-3285	-835
39	-911	-775	878	110	-495	1006	210	728	-3255	7673
40	-435	682	1077	223	1909	-2036	2401	-1488	170	146
41	-1198	-729	200	1756	657	-562	2930	-1896	-760	569
42	-402	-833	-1728	-1750	-410	1849	1699	-567	594	-1553
43	-2410	-992	-784	-55	181	206	2038	-1294	-5254	1128
44	183	-1304	-251	-43	-1774	588	3407	-6621	-2567	-811
45	-968	-765	530	-424	-1	916	945	-1418	-215	-11098
46	-387	-813	-507	-3149	-587	-334	2211	953	-177	2276
47	-4723	-2206	387	-2062	1889	242	-460	295	-50	1006
48	148	-1071	5072	591	-2189	271	-891	1074	1059	-269
49	610	-1281	1949	2982	-300	2178	1236	-1529	1117	2360
50	-964	-1344	808	772	218	4011	1237	11201	5525	6017
51	-1284	-902	-1566	935	1881	2420	2133	1300	-508	-605
52	253	-1230	1713	-143	-4161	-46	972	-3265	2170	-1523
53	293	-985	224	1218	-1262	392	-21	-1539	3888	-4842
54	1135	777	142	-1582	-2476	2111	752	-1979	-819	-3038
55	-1636	-3260	-2040	1427	-1727	1199	704	-383	-1769	-1835
56	84	997	466	1081	-699	-424	-1982	-4373	5391	-2342
57	927	1404	173	1332	715	1537	4062	-3081	-2540	265
58	-469	986	610	-1306	-366	2774	955	1021	1785	4607
59	-45	811	-1953	1637	1037	126	376	-1108	-3463	339
60	-466	1024	-256	-234	-559	-601	2253	-4365	7628	2666
61	250	-820	-1061	-430	-1	-1248	2152	-2411	3407	-9281
62	-923	746	-980	-318	-1947	-611	-1862	-41	2343	4071
63	-1280	-879	-1667	1035	-453	-426	-725	-3454	674	1283
64	2634	1214	3448	394	-462	228	382	-1273	-558	-1341
65	1627	814	332	1085	527	977	-1435	-625	-2965	860
66	656	770	389	-1721	1732	761	140	-1008	-3829	-5355
67	-206	2713	263	421	-59	-440	477	593	616	576
68	595	-700	-107	-1406	-1509	-2131	-50	341	143	4214
69	2611	-1155	-866	1679	-1	1544	-692	2680	893	-1559
70	758	793	-1172	-1539	-2007	-646	323	30	2017	4193
71	2312	981	-173	653	-1087	-1344	2133	-650	28	968
72	-318	999	2863	-1696	307	681	-926	-1649	-39	-781
73	1284	799	891	-801	1564	-2095	-1583	848	-1212	-327
74	1808	1162	66	-4747	3304	1008	-173	-423	-1215	1181
75	2641	1069	-902	232	2953	688	522	268	293	175
76	797	-1150	1524	-4061	3382	-287	-484	-45	-3144	-3137
77	470	-666	-106	-1850	1222	-829	-293	1278	1657	-5862
78	-1069	-1820	-3524	-5746	2775	-406	-445	-44	1401	1840
79	1027	831	-346	950	1179	-982	1914	222	1854	2065
80	-129	733	1895	58	-1454	746	1360	1403	-1829	-3023
81	300	901	1115	2174	-863	-1753	632	6038	-2705	-874
82	-722	886	485	-1167	-214	1057	-3351	2192	-5680	-3009
83	-824	5704	-1072	2166	346	-335	528	304	-376	-858
84	1073	1305	1095	269	-3150	618	-826	2718	2248	6336
85	979	-1038	-1193	-271	-1048	-620	29	5764	4654	2798

86	201	-729	-2020	-1455	-100	-444	-1882	1077	-3250	-928
87	-571	950	-1500	130	-763	1072	-1895	4771	-1534	-3396
88	-158	875	465	-1906	441	72	-1897	-3564	1026	4080
89	-20	982	-334	-101	2873	-544	-2399	-1914	-1003	-1133
90	-1045	729	-1377	-2187	874	2318	-872	-52	-1335	-1465
91	-697	2235	-2466	-177	525	1147	1289	-2700	-424	86
92	697	1379	1799	642	14	-1276	-215	2441	16386	10749
93	-554	-794	-418	-40	1042	-382	-1777	7380	618	-1064
94	1822	1848	-3881	-582	-1192	1700	-540	639	4302	3165
95	166	793	-1864	-171	1504	-175	-1231	-792	3369	1618
96	87	1121	-1578	-1136	-2467	-2734	-7351	-1152	-2241	-8569
97	855	970	-420	301	36	582	-8104	426	-8356	-9449
98	952	1536	432	-2641	-115	-1466	5846	233	-1257	-2578
99	-2224	1399	-1103	-450	1763	-998	1686	1920	-736	-47
100	-50	1189	-1158	-1160	-3325	-15702	-2440	-887	-4671	-1783
101	183	-828	-1639	626	237	395	-2563	167	-1248	-11786
102	-87	841	-985	605	-2211	-1697	5499	238	127	4349
103	-1165	1051	-848	-402	-1601	-332	2467	252	-3375	847
104	253	-1060	417	413	68	-1570	-3920	-10768	1361	-5063
105	-166	-1070	-193	-170	111	-740	-3604	-2427	-11876	-5145
106	-10	616	-659	-1902	294	244	2431	-33	1045	-1334
107	-1057	-726	-302	-1061	1595	-1635	-514	-511	-2395	266
108	-940	-1624	-582	-786	-468	-4660	-1003	-15749	-6529	-2107
109	97	-1067	63	-165	-1211	-1610	224	-1875	-3150	-19091
110	217	-1432	-2153	-1075	833	-1807	3821	-1762	-1981	-121
111	209	1024	-535	-1537	1524	-3995	3431	-1481	2876	-237
112	30	-1606	551	1401	-4167	-1196	-2925	613	-2783	-3431
113	-221	1002	-928	526	-1723	1525	-2327	-2027	-2499	-607
114	-2516	1081	-673	1540	33	-33	-1258	1336	-2577	-223
115	-5703	2336	-1985	1779	197	370	931	-409	540	-540
116	438	1551	-1680	-902	-8699	-284	-91	-1765	-2392	-1349
117	581	1066	-308	265	-1940	72	-584	-344	564	-10666
118	-461	-1095	-3661	-587	-3356	-1699	647	-1080	2132	24
119	-2050	1114	-2993	-786	-1373	-398	258	-373	3718	-35
120	741	1008	-744	-1369	-1576	-2146	-7186	-21217	-308	-2551
121	243	949	-942	-268	122	817	1439	-2225	-15141	-4756
122	-1657	788	16	-902	-1171	1864	324	-2561	397	-3813
123	-819	706	-1349	-35	-290	-2124	-637	-2343	-1058	-890
124	-956	-2053	-1641	-1195	-1850	-4251	-10855	-15001	3517	-3817
125	-417	1287	964	-355	359	-935	-2955	-1426	289	-14426
126	1541	-786	-2131	-433	-802	-932	-2248	-2316	4924	96
127	202	1178	-2964	833	-995	-5359	69	-4858	1256	-4349
128	1488	-1955	-6590	1047	422	567	311	-46	916	397
129	-445	-1443	-1785	-6945	-2534	-806	-2409	-1184	-707	-4339
130	-553	-2225	-250	-390	-3316	-4497	67	2833	169	-3250
131	-1219	-1234	-1126	-639	-1238	-1503	-5514	-4088	-258	-4326
132	716	923	-1856	2189	-1003	851	1693	-2519	3541	1953
133	-3885	4033	1171	-1778	-1840	201	-423	254	1660	-110
134	-532	674	-26	1549	-531	-896	1359	2189	817	-2616
135	-1160	652	444	-879	264	-1034	-1506	637	759	-5774
136	284	767	-2894	-163	-463	-532	1775	1743	-1438	-3535
137	-14	-1248	-1582	-1472	-7825	41	1292	738	-638	-2979
138	-1723	1233	729	2381	-2160	-951	366	-2293	-2087	-2766
139	-633	1517	1386	104	-4742	-1236	553	-2077	1963	19
140	889	1170	-1810	2026	-1300	-1412	-2235	2597	365	73
141	315	1085	-1493	804	-4255	-581	384	1215	-4135	-1961
142	367	1149	-843	3832	751	-1773	141	-1380	-2936	-1253

143	1436	2531	1351	-2589	-3187	-548	322	-1020	-2293	-235
144	3201	3223	-7383	-297	543	-651	678	323	-1849	-103
145	-279	953	88	-4524	-1394	-985	368	162	-3737	-869
146	-140	-1243	-1575	-1246	-1088	-12797	2644	-1357	-1423	-3826
147	-276	-1147	496	-2601	-816	-5761	409	-2353	-180	-3808
148	1839	-850	-1755	-987	1721	136	1827	-1149	-1978	-2236
149	1593	989	-2006	-1295	178	-383	-370	-2375	-2045	1645
150	-676	-706	-442	406	-582	-3248	2396	1511	-102	834
151	-958	-743	1318	-380	-143	-1849	-618	-2813	-646	691
152	1454	1321	-534	226	169	-498	728	3274	-6877	-4855
153	2038	-907	-756	-503	-2000	682	511	1216	-455	-2455
154	-654	1055	-80	932	-1317	-7334	623	987	-2224	-1816
155	1838	-934	1394	-1145	-1012	-1086	-30	1165	-3178	-5932
156	650	1000	-439	-28	-855	-1335	-309	-3629	-8164	-22246
157	74	-624	-981	277	-2096	-949	252	-1934	-2831	-1794
158	390	824	-10	502	-180	-1024	3594	-1312	-453	-13048
159	441	689	748	-339	-1238	-1602	256	-1273	-4056	-1139
160	488	-847	-1551	1291	1350	2360	-1121	-1975	-2557	-1389
161	28	1340	598	-4263	-267	583	-339	-2073	5695	1087
162	-604	1471	121	810	204	-556	-8401	-9515	1066	-529
163	990	-1082	143	-2570	-2955	-846	-3306	-2630	2567	953
164	-612	870	-816	2482	2483	141	-219	1016	8297	4247
165	-1846	970	555	-1981	-70	-36	506	3256	4191	3182
166	-277	1461	-1531	1314	77	841	-5243	549	3285	835
167	911	775	-878	-110	495	-1006	-210	-728	3255	-7673
168	435	-682	-1077	-223	-1909	2036	-2401	1488	-170	-146
169	1198	729	-200	-1756	-657	562	-2930	1896	760	-569
170	402	833	1728	1750	410	-1849	-1699	567	-594	1553
171	2410	992	784	55	-181	-206	-2038	1294	5254	-1128
172	-183	1304	251	43	1774	-588	-3407	6621	2567	811
173	968	765	-530	424	1	-916	-945	1418	215	11098
174	387	813	507	3149	587	334	-2211	-953	177	-2276
175	4723	2206	-387	2062	-1889	-242	460	-295	50	-1006
176	-148	1071	-5072	-591	2189	-271	891	-1074	-1059	269
177	-610	1281	-1949	-2982	300	-2178	-1236	1529	-1117	-2360
178	964	1344	-808	-772	-218	-4011	-1237	-11201	-5525	-6017
179	1284	902	1566	-935	-1881	-2420	-2133	-1300	508	605
180	-253	1230	-1713	143	4161	46	-972	3265	-2170	1523
181	-293	985	-224	-1218	1262	-392	21	1539	-3888	4842
182	-1135	-777	-142	1582	2476	-2111	-752	1979	819	3038
183	1636	3260	2040	-1427	1727	-1199	-704	383	1769	1835
184	-84	-997	-466	-1081	699	424	1982	4373	-5391	2342
185	-927	-1404	-173	-1332	-715	-1537	-4062	3081	2540	-265
186	469	-986	-610	1306	366	-2774	-955	-1021	-1785	-4607
187	45	-811	1953	-1637	-1037	-126	-376	1108	3463	-339
188	466	-1024	256	234	559	601	-2253	4365	-7628	-2666
189	-250	820	1061	430	1	1248	-2152	2411	-3407	9281
190	923	-746	980	318	1947	611	1862	41	-2343	-4071
191	1280	879	1667	-1035	453	426	725	3454	-674	-1283
192	-2634	-1214	-3448	-394	462	-228	-382	1273	558	1341
193	-1627	-814	-332	-1085	-527	-977	1435	625	2965	-860
194	-656	-770	-389	1721	-1732	-761	-140	1008	3829	5355
195	206	-2713	-263	-421	59	440	-477	-593	-616	-576
196	-595	700	107	1406	1509	2131	50	-341	-143	-4214
197	-2611	1155	866	-1679	1	-1544	692	-2680	-893	1559
198	-758	-793	1172	1539	2007	646	-323	-30	-2017	-4193
199	-2312	-981	173	-653	1087	1344	-2133	650	-28	-968

200	318	-999	-2863	1696	-307	-681	926	1649	39	781
201	-1284	-799	-891	801	-1564	2095	1583	-848	1212	327
202	-1808	-1162	-66	4747	-3304	-1008	173	423	1215	-1181
203	-2641	-1069	902	-232	-2953	-688	-522	-268	-293	-175
204	-797	1150	-1524	4061	-3382	287	484	45	3144	3137
205	-470	666	106	1850	-1222	829	293	-1278	-1657	5862
206	1069	1820	3524	5746	-2775	406	445	44	-1401	-1840
207	-1027	-831	346	-950	-1179	982	-1914	-222	-1854	-2065
208	129	-733	-1895	-58	1454	-746	-1360	-1403	1829	3023
209	-300	-901	-1115	-2174	863	1753	-632	-6038	2705	874
210	722	-886	-485	1167	214	-1057	3351	-2192	5680	3009
211	824	-5704	1072	-2166	-346	335	-528	-304	376	858
212	-1073	-1305	-1095	-269	3150	-618	826	-2718	-2248	-6336
213	-979	1038	1193	271	1048	620	-29	-5764	-4654	-2798
214	-201	729	2020	1455	100	444	1882	-1077	3250	928
215	571	-950	1500	-130	763	-1072	1895	-4771	1534	3396
216	158	-875	-465	1906	-441	-72	1897	3564	-1026	-4080
217	20	-982	334	101	-2873	544	2399	1914	1003	1133
218	1045	-729	1377	2187	-874	-2318	872	52	1335	1465
219	697	-2235	2466	177	-525	-1147	-1289	2700	424	-86
220	-697	-1379	-1799	-642	-14	1276	215	-2441	-16386	-10749
221	554	794	418	40	-1042	382	1777	-7380	-618	1064
222	-1822	-1848	3881	582	1192	-1700	540	-639	-4302	-3165
223	-166	-793	1864	171	-1504	175	1231	792	-3369	-1618
224	-87	-1121	1578	1136	2467	2734	7351	1152	2241	8569
225	-855	-970	420	-301	-36	-582	8104	-426	8356	9449
226	-952	-1536	-432	2641	115	1466	-5846	-233	1257	2578
227	2224	-1399	1103	450	-1763	998	-1686	-1920	736	47
228	50	-1189	1158	1160	3325	15702	2440	887	4671	1783
229	-183	828	1639	-626	-237	-395	2563	-167	1248	11786
230	87	-841	985	-605	2211	1697	-5499	-238	-127	-4349
231	1165	-1051	848	402	1601	332	-2467	-252	3375	-847
232	-253	1060	-417	-413	-68	1570	3920	10768	-1361	5063
233	166	1070	193	170	-111	740	3604	2427	11876	5145
234	10	-616	659	1902	-294	-244	-2431	33	-1045	1334
235	1057	726	302	1061	-1595	1635	514	511	2395	-266
236	940	1624	582	786	468	4660	1003	15749	6529	2107
237	-97	1067	-63	165	1211	1610	-224	1875	3150	19091
238	-217	1432	2153	1075	-833	1807	-3821	1762	1981	121
239	-209	-1024	535	1537	-1524	3995	-3431	1481	-2876	237
240	-30	1606	-551	-1401	4167	1196	2925	-613	2783	3431
241	221	-1002	928	-526	1723	-1525	2327	2027	2499	607
242	2516	-1081	673	-1540	-33	33	1258	-1336	2577	223
243	5703	-2336	1985	-1779	-197	-370	-931	409	-540	540
244	-438	-1551	1680	902	8699	284	91	1765	2392	1349
245	-581	-1066	308	-265	1940	-72	584	344	-564	10666
246	461	1095	3661	587	3356	1699	-647	1080	-2132	-24
247	2050	-1114	2993	786	1373	398	-258	373	-3718	35
248	-741	-1008	744	1369	1576	2146	7186	21217	308	2551
249	-243	-949	942	268	-122	-817	-1439	2225	15141	4756
250	1657	-788	-16	902	1171	-1864	-324	2561	-397	3813
251	819	-706	1349	35	290	2124	637	2343	1058	890
252	956	2053	1641	1195	1850	4251	10855	15001	-3517	3817
253	417	-1287	-964	355	-359	935	2955	1426	-289	14426
254	-1541	786	2131	433	802	932	2248	2316	-4924	-96
255	-202	-1178	2964	-833	995	5359	-69	4858	-1256	4349

Contents for Subpart 3

Subpart 3: Speech Coding - CELP

3.1 Scope

3.1.1 General description of the CELP decoder

This subclause provides a brief overview of the CELP (Code Excited Linear Prediction) decoder. A basic block diagram of the CELP decoder is given in Figure 3.1.

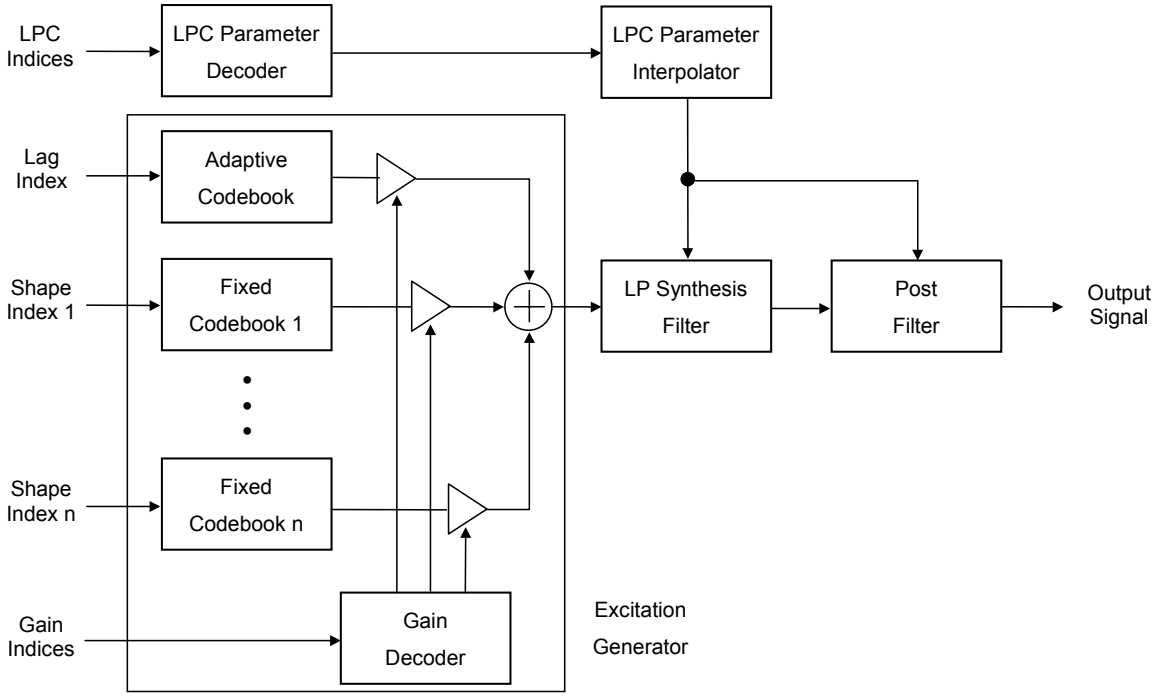


Figure 3.1 — Block diagram of a CELP decoder

The CELP decoder primarily consists of an excitation generator and a synthesis filter. Additionally, CELP decoders often include a post-filter. The excitation generator has an adaptive codebook to model periodic components, fixed codebooks to model random components and a gain decoder to represent a speech signal level. Indices for the codebooks and gains are provided by the encoder. The codebook indices (pitch-lag index for the adaptive codebook and shape index for the fixed codebook) and gain indices (adaptive and fixed codebook gains) are used to generate the excitation signal. It is then filtered by the linear predictive synthesis filter (LP synthesis filter). Filter coefficients are reconstructed using the LPC indices, then are interpolated with the filter coefficients of successive analysis frames. Finally, a post-filter can optionally be applied in order to enhance the speech quality.

3.1.2 Functionality of MPEG-4 CELP

MPEG-4 CELP is a generic coding algorithm with new functionalities. Conventional CELP coders offer compression at a single bitrate and are optimized for specific applications. Compression is one of the functions provided by MPEG-4 CELP, enabling the use of one basic coder for various applications. It provides scalability in bitrate and bandwidth, as well as the ability to generate bitstreams at arbitrary bitrates. The MPEG-4 CELP coder supports two sampling rates, namely, 8 and 16 kHz. The associated bandwidths are 100 – 3400 Hz for 8 kHz sampling rate and 50 – 7000 Hz for 16 kHz sampling rate. Furthermore, silence compression and error resilient bitstream reordering are newly adopted.

3.1.2.1 Configuration of the MPEG-4 CELP coder

Two different tools can be used to generate the excitation signal. These are the Multi-Pulse Excitation (MPE) tool or the Regular-Pulse Excitation (RPE) tool. MPE is used for speech sampled at 8 kHz or 16 kHz. RPE is only used for speech sampled at 16 kHz. The two possible coding modes are summarized in Table 3.1.

Table 3.1 — Coding modes in the MPEG-4 CELP coder

Coding Mode	Excitation tool	Sampling rate
I	RPE	16 kHz
II	MPE	8, 16 kHz

3.1.2.2 Features of the MPEG-4 CELP coder

The MPEG-4 CELP coder offers the following functionality, depending on the coding mode.

Table 3.2 — Functionality of the MPEG-4 CELP coder

Coding Mode	Functionality
I	Multiple bitrates, FineRate Control
II	Multiple bitrates, Bitrate Scalability, Bandwidth Scalability, FineRate Control

For both coding modes, silence compression and error resilient bitstream reordering are available.

Multiple bitrates: The available bitrates depend on the coding mode and the sampling rate. The following fixed bitrates are supported:

Table 3.3 — Fixed bitrates for the mode I coder

Bitrates for the 16 kHz sampling rate (bit/s)
14400, 16000, 18667, 22533

Table 3.4 — Fixed bitrates for the mode II coder

Bitrates for the 8 kHz sampling rate (bit/s)	Bitrates for the 16 kHz sampling rate (bit/s)
3850, 4250, 4650, 4900, 5200, 5500, 5700, 6000, 6200, 6300, 6600, 6900, 7100, 7300, 7700, 8300, 8700, 9100, 9500, 9900, 10300, 10500, 10700, 11000, 11400, 11800, 12000, 12200	10900, 11500, 12100, 12700, 13300, 13900, 14300, 14700, 15900, 17100, 17900, 18700, 19500, 20300, 21100, 13600, 14200, 14800, 15400, 16000, 16600, 17000, 17400, 18600, 19800, 20600, 21400, 22200, 23000, 23800

During non-active frames, the silence compression tool is used and the CELP coder operates at bitrates shown in Table 3.5. The bitrate depends on the coding mode, the sampling rate and the frame length.

Table 3.5 — Bitrates for the silence compression tool

Coding mode	Sampling rate [kHz]	Band width scalability	Frame length [ms]	Bitrate [bit/s]		
				TX_flag	HR-SID	LR-SID
I (RPE)	16	-	15	133	2533	400
			10	200	3800	600
II (MPE)	8	On, Off	40	50	525	150
			30	67	700	200
			20	100	1050	300
			10	200	2100	600
	16	Off	20	100	1900	300
			10	200	3800	600
		On	40	50	1050	150
			30	67	1400	200
			20	100	2100	300
			10	200	4200	600

Fine Rate Control: Enables fine step bitrate control (permitting variable bitrate operation). This is achieved purely by controlling the transmission rate of the LPC parameters using a combinations of the two bitstream elements **interpolation_flag** and **LPC_present** flag. Using FineRate Control it is possible to vary the ratio of LPC-frames to total frames between 50% and 100%. This enables the bitrate to be decreased with respect to the anchor bitrate, as defined in the Semantics.

Bitrate Scalability: Bitrate scalability is provided by adding enhancement layers. Enhancement layers can be added with a step of 2000 bit/s for signals sampled at 8 kHz or 4000 bit/s for signals sampled at 16 kHz. A maximum of three enhancement layers may be combined with any bitrate chosen from Table 3.4.

Bandwidth Scalability: Bandwidth scalability to cover both sampling rates is achieved by incorporating a bandwidth extension tool in the CELP coder. This is an enhancement tool, supported in Mode II, which may be added if scalability from the 8 kHz sampling rate to the 16 kHz sampling rate is required. A complete coder with bandwidth scalability consists of a core CELP coder for the 8 kHz sampling rate and the bandwidth extension tool to provide a single layer of scalability. The core CELP coder for the 8 kHz sampling rate can comprise several layers. It should be noted that an 8 kHz sampling rate coder with this tool is not the same as a 16 kHz sampling rate coder. Both configurations (8 kHz sampling rate coder with bandwidth scalability and 16 kHz sampling rate coder) offer greater intelligibility and naturalness of decoded speech than does the 8 kHz coder alone because they expand the bandwidth to 7 kHz. The additional bitrate required for the bandwidth scalability tool can be selected from 4 discrete steps for each core layer bitrate as shown in Table 3.6.

Table 3.6 — Bitrates for the bandwidth scalable mode

Bitrate of the core layer (bit/s)	Additional bitrate (bit/s)
3850 - 4650	+9200, +10400, +11600, +12400
4900 - 5500	+9467, +10667, +11867, +12667
5700 - 10700	+10000, +11200, +12400, +13200
11000 - 12200	+11600, +12800, +14000, +14800

Silence compression: The silence compression tool can be used to reduce the bitrate for input signals with little voice activity. For such non-active periods, the decoder substitutes the regular excitation signal with synthetically generated noise. For voice active periods, the regular speech synthesis process is always used. The silence compression tool is available when the ER-CELP object type is used.

Error resilient bitstream reordering: Error resilient bitstream reordering allows the effective use of advanced channel coding techniques like unequal error protection (UEP). The basic idea is to rearrange the audio frame content depending on its error sensitivity in one or more instances belonging to different error sensitivity categories (ESC). This rearrangement works either data element-wise or even bit-wise. An error resilient bitstream frame is build by concatenating these instances. This functionality is available when the ER-CELP object type is used.

3.1.2.3 Algorithmic delay of MPEG-4 CELP modes

The algorithmic delay of the CELP coder comes from the frame length and an additional look ahead length. The frame length depends on the coding mode and the bitrate. The look ahead length, which is an informative parameter, also depends on the coding mode. The delays presented below are applicable to the modes where FineRate Control is off. When FineRate Control is on, additional one-frame delay is introduced. Bandwidth scalability in the mode II coder requires an additional look ahead of 5 ms due to down-sampling.

Table 3.7 — Delay and frame length for the mode I coder of the 16 kHz sampling rate

Bitrate for Mode I (bit/s)	Delay (ms)	Frame Length (ms)
14400	26.25	15
16000	18.75	10
18667	26.56	15
22533	26.75	15

Table 3.8 — Delay and frame length for the mode II coder of the 8 kHz sampling rate

Bitrate for Mode II (bit/s)	Delay (ms)	Frame Length (ms)
3850, 4250, 4650	45	40
4900, 5200, 5500, 6200	35	30
5700, 6000, 6300, 6600, 6900, 7100, 7300, 7700, 8300, 8700, 9100, 9500, 9900, 10300, 10500, 10700	25	20
11000, 11400, 11800, 12000, 12200	15	10

Table 3.9 — Delay and frame length for the mode II coder of the 16 kHz sampling rate

Bitrate for Mode II (bit/s)	Delay (ms)	Frame Length (ms)
10900, 11500, 12100, 12700, 13300, 13900, 14300, 14700, 15900, 17100, 17900, 18700, 19500, 20300, 21100	25	20
13600, 14200, 14800, 15400, 16000, 16600, 17000, 17400, 18600, 19800, 20600, 21400, 22200, 23000, 23800	15	10

In case silence compression is used, the algorithmic delay is the same as without silence compression, since the same frame length and the same additional look-ahead length are used.

3.2 Definitions

Definitions can be found in subpart 1, subclause 1.3.

3.3 Bitstream syntax

3.3.1 CELP object type

3.3.1.1 Header syntax

CelpSpecificConfig()

The following CelpSpecificConfig() is required for the CELP object type:

Table 3.10 — Syntax of CelpSpecificConfig ()

Syntax	No. of bits	Mnemonic
CelpSpecificConfig (uint(4) samplingFrequencyIndex)		
{		
isBaseLayer	**1**	**uimsbf**
if (isBaseLayer)		
{		
CelpHeader (samplingFrequencyIndex);		
}		
else		
{		
isBWSLayer;	**1**	**uimsbf**
if (isBWSLayer)		
{		
CelpBWSenhHeader ();		
}		
else		
{		
CELP-BRS-id;	**2**	**uimsbf**
}		
}		
}		

Table 3.11 — Syntax of CelpHeader()

Syntax	No. of bits	Mnemonic
CelpHeader (samplingFrequencyIndex)		
{		
ExcitationMode;	**1**	**uimsbf**
SampleRateMode;	**1**	**uimsbf**
FineRateControl;	**1**	**uimsbf**
if (ExcitationMode == RPE) {		
RPE_Configuration;	**3**	**uimsbf**
}		
if (ExcitationMode == MPE) {		
MPE_Configuration;	**5**	**uimsbf**
NumEnhLayers;	**2**	**uimsbf**
BandwidthScalabilityMode;	**1**	**uimsbf**
}		
}		

Table 3.12 — Syntax of CelpBWSenhHeader()

Syntax	No. of bits	Mnemonic
CelpBWSenhHeader ()		
{		
BWS_configuration;	**2**	**uimsbf**
}		

3.3.1.2 Frame syntax

Transmission of CELP bitstreams

Each layer of an MPEG-4 CELP audio bitstream is transmitted in an Elementary Stream. In slPacketPayload, the following dynamic data for CELP Audio has to be included:

CELP Base Layer -- Access Unit payload

```
slPacketPayload
{
    CelpBaseFrame();
}
```

CELP Enhancement Layer -- Access Unit payload

To parse and decode the CELP enhancement layer, information decoded from the CELP base layer is required. For the bitrate scalable mode, the following data for the CELP enhancement layer has to be included:

```
slPacketPayload
{
    CelpBRSenhFrame();
}
```

For the bandwidth scalable mode, the following data for the CELP enhancement layer has to be included:

```
slPacketPayload
{
    CelpBWSenhFrame();
}
```

In case bitrate scalability and bandwidth scalability are both used simultaneously, first all bitrate enhancement layers have to be conveyed prior to the bandwidth scalability layer.

Table 3.13 — Syntax of CelpBaseFrame()

Syntax	No. of bits	Mnemonic

CelpBaseFrame()
{
Celp_LPC();
if (ExcitationMode == MPE) {
MPE_frame();
}
if ((ExcitationMode == RPE) && (SampleRateMode == 16kHz))
{
RPE_frame();
}
}		

Table 3.14 — Syntax of CelpBRSenhFrame()

Syntax	No. of bits	Mnemonic
CelpBRSenhFrame()		
{		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_enh_positions [subframe][enh_layer];	**4, 12**	**uimsbf**
shape_enh_signs [subframe][enh_layer];	**2, 4**	**uimsbf**
gain_enh_index [subframe][enh_layer];	**4**	**uimsbf**
}		
}		

Table 3.15 — Syntax of CelpBWSenhFrame()

Syntax	No. of bits	Mnemonic
CelpBWSenhFrame()		
{		
BandScalable_LSP()		
for (subframe = 0; subframe < nrof_subframe_bws; subframe++)		
{		
shape_bws_delay [subframe];	**3**	**uimsbf**
shape_bws_positions [subframe];	**22, 26, 30, 32**	**uimsbf**
shape_bws_signs [subframe];	**6, 8, 10, 12**	**uimsbf**
gain_bws_index [subframe];	**11**	**uimsbf**
}		
}		

3.3.1.2.1 LPC syntax

Table 3.16 — Syntax of Celp_LPC()

Syntax	No. of bits	Mnemonic
Celp_LPC()		
{		
if (FineRateControl == ON){		
interpolation_flag;	**1**	**uimsbf**
LPC_Present;	**1**	**uimsbf**
if (LPC_Present == YES) {		
LSP_VQ();		
}		
} else {		
LSP_VQ();		
}		
}		

Table 3.17 — Syntax of LSP_VQ()

Syntax	No. of bits	Mnemonic
LSP_VQ()		
{		
if (SampleRateMode == 8kHz) {		
NarrowBand_LSP();		
} else {		
WideBand_LSP();		
}		
}		

Table 3.18 — Syntax of NarrowBand_LSP()

Syntax	No. of bits	Mnemonic
NarrowBand_LSP()		
{		
lpc_indices [0];	**4**	**uimsbf**
lpc_indices [1];	**4**	**uimsbf**
lpc_indices [2];	**7**	**uimsbf**
lpc_indices [3];	**6**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
}		

Table 3.19 — Syntax of BandScalable_LSP()

Syntax	No. of bits	Mnemonic
BandScalable_LSP()		
{		
lpc_indices [5];	**4**	**uimsbf**
lpc_indices [6];	**7**	**uimsbf**
lpc_indices [7];	**4**	**uimsbf**
lpc_indices [8];	**6**	**uimsbf**
lpc_indices [9];	**7**	**uimsbf**
lpc_indices [10];	**4**	**uimsbf**
}		

Table 3.20 — Syntax of WideBand_LSP()

Syntax	No. of bits	Mnemonic
WideBand_LSP()		
{		
lpc_indices [0];	**5**	**uimsbf**
lpc_indices [1];	**5**	**uimsbf**
lpc_indices [2];	**7**	**uimsbf**
lpc_indices [3];	**7**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
lpc_indices [5];	**4**	**uimsbf**
lpc_indices [6];	**4**	**uimsbf**
lpc_indices [7];	**7**	**uimsbf**
lpc_indices [8];	**5**	**uimsbf**
lpc_indices [9];	**1**	**uimsbf**
}		

3.3.1.2.2 Excitation syntax

Table 3.21 — Syntax of RPE_frame()

Syntax	No. of bits	Mnemonic
RPE_frame()		
{		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay [subframe];	**8**	**uimsbf**
shape_index [subframe];	**11,12**	**uimsbf**
gain_indices [0][subframe];	**6**	**uimsbf**
gain_indices [1][subframe];	**3,5**	**uimsbf**
}		
}		

Table 3.22 — Syntax of MPE_frame()

Syntax	No. of bits	Mnemonic
MPE_frame()		
{		
signal_mode;	**2**	**uimsbf**
rms_index;	**6**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay [subframe];	**8, 9**	**uimsbf**
shape_positions [subframe];	**14 ... 32**	**uimsbf**
shape_signs [subframe];	**3 ... 12**	**uimsbf**
gain_index [subframe];	**6, 7**	**uimsbf**
}		
}		

3.3.2 ER-CELP object type

3.3.2.1 Header syntax

ErrorResilientCelpSpecificConfig()

The following ErrorResilientCelpSpecificConfig () is required for the ER-CELP object type:

Table 3.23 — Syntax of ErrorResilientCelpSpecificConfig ()

Syntax	No. of bits	Mnemonic
ErrorResilientCelpSpecificConfig (uint(4) samplingFrequencyIndex)		
{		
isBaseLayer;	**1**	**uimsbf**
if (isBaseLayer)		
{		
ER_SC_CelpHeader (samplingFrequencyIndex);		
}		
else		
{		
isBWSLayer;	**1**	**uimsbf**
if (isBWSLayer)		
{		
CelpBWSenhHeader ();		
}		
else		
{		
CELP-BRS-id;	**2**	**uimsbf**
}		
}		
}		

Table 3.24 — Syntax of ER_SC_CelpHeader()

Syntax	No. of bits	Mnemonic
ER_SC_CelpHeader (samplingFrequencyIndex)		
{		
ExcitationMode;	**1**	**uimsbf**
SampleRateMode;	**1**	**uimsbf**
FineRateControl;	**1**	**uimsbf**
SilenceCompression;	**1**	**uimsbf**
if (ExcitationMode == RPE) {		
RPE_Configuration;	**3**	**uimsbf**
}		
if (ExcitationMode == MPE) {		
MPE_Configuration;	**5**	**uimsbf**
NumEnhLayers;	**2**	**uimsbf**
BandwidthScalabilityMode;	**1**	**uimsbf**
}		
}		

3.3.2.2 Frame syntax

In order to describe the bit error sensitivity of bitstream elements, error sensitivity categories (ESC) are introduced. To describe single bits of elements, the following notation is used.

gain, x-y

Denotes bit x to bit y of element gain, whereby x is transmitted first. The LSB is bit zero and the MSB of an element that consist of N bit is N-1. The MSB is always the first bit in the bitstream.

The following syntax is a replacement for CelpBaseFrame. The syntax for enhancement layer for bitrate and bandwidth scalability is not affected.

Transmission of CELP bitstreams

The payload data for the ER CELP object is transmitted as slPacketPayload payload in the base layer and the optional enhancement layer Elementary Stream.

Error Resilient CELP Base Layer -- Access Unit payload

```
slPacketPayload
{
    ER_SC_CelpBaseFrame();
}
```

Error Resilient CELP Enhancement Layer -- Access Unit payload

To parse and decode the Error Resilient CELP enhancement layers, information decoded from the Error Resilient CELP base layer is required. For the bitrate scalable mode, the following data for the Error Resilient CELP enhancement layers has to be included:

```
slPacketPayload
{
    ER_SC_CelpBRSenhFrame();
}
```

For the bandwidth scalable mode, the following data for the Error Resilient CELP enhancement layer has to be included:

```
slPacketPayload
{
    ER_SC_CelpBWSenhFrame();
}
```

3.3.2.2.1 CELP base layer

Table 3.25 — Syntax of ER_SC_CelpBaseFrame()

Syntax	No. of bits	Mnemonic

ER_SC_CelpBaseFrame()
{
  if (SilenceCompression == OFF) {
  ER_CelpBaseFrame();
  } else {
  SC_VoiceActivity_ESC0();
  if (TX_flag == 1) {
  ER_CelpBaseFrame();
  } else if (TX_flag == 2) {
  SID_LSP_VQ_ESC0();
  SID_Frame_ESC0();
  } else if (TX_flag == 3) {
  SID_Frame_ESC0();
  }
  }
}		

Table 3.26 — Syntax of SC_VoiceActivity_ESC0 ()

Syntax	No. of bits	Mnemonic
SC_VoiceActivity_ESC0 ()		
{		
TX_flag;	**2**	**uimsbf**
}		

Table 3.27 — Syntax of ER_CelpBaseFrame ()

Syntax	No. of bits	Mnemonic
ER_CelpBaseFrame()		
{		
if (ExcitationMode == MPE) {		
if (SampleRateMode == 8kHz) {		
MPE_NarrowBand_ESC0();		
MPE_NarrowBand_ESC1();		
MPE_NarrowBand_ESC2();		
MPE_NarrowBand_ESC3();		
MPE_NarrowBand_ESC4();		
}		
if (SampleRateMode == 16kHz) {		
MPE_WideBand_ESC0();		
MPE_WideBand_ESC1();		
MPE_WideBand_ESC2();		
MPE_WideBand_ESC3();		
MPE_WideBand_ESC4();		
}		
}		
if ((ExcitationMode == RPE) && (SampleRateMode == 16kHz)) {		
RPE_WideBand_ESC0();		
RPE_WideBand_ESC1();		
RPE_WideBand_ESC2();		
RPE_WideBand_ESC3();		
RPE_WideBand_ESC4();		
}		
}		

3.3.2.2.1.1 MPE narrowband syntax

Table 3.28 — Syntax of MPE_NarrowBand_ESC0()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC0()		
{		
if (FineRateControl == ON) {		
interpolation_flag;	**1**	**uimsbf**
LPC_Present;	**1**	**uimsbf**
}		
rms_index, 5-4;	**2**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 7;	**1**	**uimsbf**
}		
}		

Table 3.29 — Syntax of MPE_NarrowBand_ESC1()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC1()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [0], 1-0;	**2**	**uimsbf**
lpc_indices [1], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [0], 1-0;	**2**	**uimsbf**
lpc_indices [1], 0;	**1**	**uimsbf**
}		
signal_mode;	**2**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 6-5;	**2**	**uimsbf**
}		
}		

Table 3.30 — Syntax of MPE_NarrowBand_ESC2()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC2()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 0;	**1**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
}		
} else {		
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 0;	**1**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
}		
rms_index, 3	**1**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 4-3;	**2**	**uimsbf**
gain_index[subframe], 1-0;	**2**	**uimsbf**
}		
}		

Table 3.31 — Syntax of MPE_NarrowBand_ESC3()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC3()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [0], 3-2;	**2**	**uimsbf**
lpc_indices [1], 2-1;	**2**	**uimsbf**
lpc_indices [2], 5-1;	**5**	**uimsbf**
}		
} else {		
lpc_indices [0], 3-2;	**2**	**uimsbf**
lpc_indices [1], 2-1;	**2**	**uimsbf**
lpc_indices [2], 5-1;	**5**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 2-0;	**3**	**uimsbf**
shape_signs[subframe];	**3 ... 12**	**uimsbf**
gain_index[subframe], 2;	**1**	**uimsbf**
}		
}		

Table 3.32 — Syntax of MPE_NarrowBand_ESC4()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC4()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 3;	**1**	**uimsbf**
lpc_indices [3];	**6**	**uimsbf**
}		
} else {		
lpc_indices [1], 3;	**1**	**uimsbf**
lpc_indices [3];	**6**	**uimsbf**
}		
rms_index, 2-0	**3**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_positions[subframe];	**13 ... 32**	**uimsbf**
gain_index[subframe], 5-3;	**3**	**uimsbf**
}		
}		

3.3.2.2.1.2 MPE wideband syntax

Table 3.33 — Syntax of MPE_WideBand_ESC0()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC0()		
{		
if (FineRateControl == ON) {		
interpolation_flag;	**1**	**uimsbf**
LPC_Present;	**1**	**uimsbf**
if (LPC_Present == YES) {		
lpc_indices [0];	**5**	**uimsbf**
lpc_indices [1], 1-0;	**2**	**uimsbf**
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 4-0;	**5**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
lpc_indices [5], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [0];	**5**	**uimsbf**
lpc_indices [1], 1-0;	**2**	**uimsbf**
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 4-0;	**5**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
lpc_indices [5], 0;	**1**	**uimsbf**
}		
rms_index, 4-5;	**2**	**uimsbf**
}		

Table 3.34 — Syntax of MPE_WideBand_ESC1()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC1()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 3-2;	**2**	**uimsbf**
lpc_indices [2], 5;	**1**	**uimsbf**
lpc_indices [5], 1;	**1**	**uimsbf**
lpc_indices [6], 1-0;	**2**	**uimsbf**
}		
} else {		
lpc_indices [1], 3-2;	**2**	**uimsbf**
lpc_indices [2], 5;	**1**	**uimsbf**
lpc_indices [5], 1;	**1**	**uimsbf**
lpc_indices [6], 1-0;	**2**	**uimsbf**
}		
signal_mode;	**2**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 8-6;	**3**	**uimsbf**
}		
}		

Table 3.35 — Syntax of MPE_WideBand_ESC2()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC2()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 4;	**1**	**uimsbf**
lpc_indices [3], 6;	**1**	**uimsbf**
lpc_indices [3], 1;	**1**	**uimsbf**
lpc_indices [5], 2;	**1**	**uimsbf**
lpc_indices [6], 3;	**1**	**uimsbf**
lpc_indices [7], 6;	**1**	**uimsbf**
lpc_indices [7], 4;	**1**	**uimsbf**
lpc_indices [7], 1-0;	**2**	**uimsbf**
lpc_indices [9];	**1**	**uimsbf**
}		
} else {		
lpc_indices [1], 4;	**1**	**uimsbf**
lpc_indices [3], 6;	**1**	**uimsbf**
lpc_indices [3], 1;	**1**	**uimsbf**
lpc_indices [5], 2;	**1**	**uimsbf**
lpc_indices [6], 3;	**1**	**uimsbf**
lpc_indices [7], 6;	**1**	**uimsbf**
lpc_indices [7], 4;	**1**	**uimsbf**
lpc_indices [7], 1-0;	**2**	**uimsbf**
lpc_indices [9];	**1**	**uimsbf**
}		
rms_index, 3;	**1**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 5-4;	**2**	**uimsbf**
gain_index[subframe], 1-0;	**2**	**uimsbf**
}		
}		

Table 3.36 — Syntax of MPE_WideBand_ESC3()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC3()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [3], 4-2;	**3**	**uimsbf**
lpc_indices [3], 0;	**1**	**uimsbf**
lpc_indices [5], 3;	**1**	**uimsbf**
lpc_indices [6], 2;	**1**	**uimsbf**
lpc_indices [7], 5;	**1**	**uimsbf**
lpc_indices [7], 3-2;	**2**	**uimsbf**
lpc_indices [8], 4-1;	**4**	**uimsbf**
}		
} else {		
lpc_indices [3], 4-2;	**3**	**uimsbf**
lpc_indices [3], 0;	**1**	**uimsbf**
lpc_indices [5], 3;	**1**	**uimsbf**
lpc_indices [6], 2;	**1**	**uimsbf**
lpc_indices [7], 5;	**1**	**uimsbf**
lpc_indices [7], 3-2;	**2**	**uimsbf**
lpc_indices [8], 4-1;	**4**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 3-2;	**2**	**uimsbf**
shape_signs[subframe];	**3 ... 12**	**uimsbf**
gain_index[subframe], 2;	**1**	**uimsbf**
}		
}		

Table 3.37 — Syntax of MPE_WideBand_ESC4()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC4()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [3], 5;	**1**	**uimsbf**
lpc_indices [8], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [3], 5;	**1**	**uimsbf**
lpc_indices [8], 0;	**1**	**uimsbf**
}		
rms_index, 2-0;	**3**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 1-0;	**2**	**uimsbf**
shape_positions[subframe];	**14 ... 32**	**uimsbf**
gain_index[subframe], 6-3;	**4**	**uimsbf**
}		
}		

3.3.2.2.1.3 RPE wideband syntax

Table 3.38 — Syntax of RPE_WideBand_ESC0()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC0()		
{		
if (FineRateControl == ON){		
interpolation_flag;	**1**	**uimsbf**
LPC_Present;	**1**	**uimsbf**
if (LPC_Present == YES) {		
lpc_indices [0];	**5**	**uimsbf**
lpc_indices [1], 1-0;	**2**	**uimsbf**
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 4-0;	**5**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
lpc_indices [5], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [0];	**5**	**uimsbf**
lpc_indices [1], 1-0;	**2**	**uimsbf**
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 4-0;	**5**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
lpc_indices [5], 0;	**1**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
gain_indices[0][subframe], 5-3;	**3**	**uimsbf**
if (subframe == 0) {		
gain_indices[1][subframe], 4-3;	**2**	**uimsbf**
} else {		
gain_indices[1][subframe], 2;	**1**	**uimsbf**
}		
}		
}		

Table 3.39 — Syntax of RPE_WideBand_ESC1()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC1()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 3-2;	**2**	**uimsbf**
lpc_indices [2], 5;	**1**	**uimsbf**
lpc_indices [5], 1;	**1**	**uimsbf**
lpc_indices [6], 1-0;	**2**	**uimsbf**
}		
} else {		
lpc_indices [1], 3-2;	**2**	**uimsbf**
lpc_indices [2], 5;	**1**	**uimsbf**
lpc_indices [5], 1;	**1**	**uimsbf**
lpc_indices [6], 1-0;	**2**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 7-5;	**3**	**uimsbf**
}		
}		

Table 3.40 — Syntax of RPE_WideBand_ESC2()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC2()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 4;	**1**	**uimsbf**
lpc_indices [3], 6;	**1**	**uimsbf**
lpc_indices [3], 1;	**1**	**uimsbf**
lpc_indices [5], 2;	**1**	**uimsbf**
lpc_indices [6], 3;	**1**	**uimsbf**
lpc_indices [7], 6;	**1**	**uimsbf**
lpc_indices [7], 4;	**1**	**uimsbf**
lpc_indices [7], 1-0;	**2**	**uimsbf**
lpc_indices [9];	**1**	**uimsbf**
}		
} else {		
lpc_indices [1], 4;	**1**	**uimsbf**
lpc_indices [3], 6;	**1**	**uimsbf**
lpc_indices [3], 1;	**1**	**uimsbf**
lpc_indices [5], 2;	**1**	**uimsbf**
lpc_indices [6], 3;	**1**	**uimsbf**
lpc_indices [7], 6;	**1**	**uimsbf**
lpc_indices [7], 4;	**1**	**uimsbf**
lpc_indices [7], 1-0;	**2**	**uimsbf**
lpc_indices [9];	**1**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 4-3;	**2**	**uimsbf**
gain_indices[0][subframe], 2;	**1**	**uimsbf**
if (subframe == 0) {		
gain_indices[1][subframe], 2;	**1**	**uimsbf**
} else {		
gain_indices[1][subframe], 1;	**1**	**uimsbf**
}		
}		
}		

Table 3.41 — Syntax of RPE_WideBand_ESC3()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC3()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [3], 4-2;	**3**	**uimsbf**
lpc_indices [3], 0;	**1**	**uimsbf**
lpc_indices [5], 3;	**1**	**uimsbf**
lpc_indices [6], 2;	**1**	**uimsbf**
lpc_indices [7], 5;	**1**	**uimsbf**
lpc_indices [7], 3-2;	**2**	**uimsbf**
lpc_indices [8], 4-1;	**4**	**uimsbf**
}		
} else {		
lpc_indices [3], 4-2;	**3**	**uimsbf**
lpc_indices [3], 0;	**1**	**uimsbf**
lpc_indices [5], 3;	**1**	**uimsbf**
lpc_indices [6], 2;	**1**	**uimsbf**
lpc_indices [7], 5;	**1**	**uimsbf**
lpc_indices [7], 3-2;	**2**	**uimsbf**
lpc_indices [8], 4-1;	**4**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 2-1;	**2**	**uimsbf**
}		
}		

Table 3.42 — Syntax of RPE_WideBand_ESC4()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC4()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [3], 5;	**1**	**uimsbf**
lpc_indices [8], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [3], 5;	**1**	**uimsbf**
lpc_indices [8], 0;	**1**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 0;	**1**	**uimsbf**
shape_index[subframe];	**11, 12**	**uimsbf**
gain_indices[0][subframe], 1-0;	**2**	**uimsbf**
if (subframe == 0) {		
gain_indices[1][subframe], 1-0;	**2**	**uimsbf**
} else {		
gain_indices[1][subframe], 0;	**1**	**uimsbf**
}		
}		
}		

3.3.2.2.2 CELP enhancement layers

Table 3.43 — Syntax of ER_SC_CelpBRSenhFrame()

Syntax	No. of bits	Mnemonic

ER_SC_CelpBRSenhFrame()
{
if (SilenceCompression == OFF) {
CelpBRSenhFrame();
} else if (TX_flag == 1) {
CelpBRSenhFrame();
}
}		

Table 3.44 — Syntax of ER_SC_CelpBWSenhFrame()

Syntax	No. of bits	Mnemonic

ER_SC_CelpBWSenhFrame()
{
if (SilenceCompression == OFF) {
CelpBWSenhFrame();
} else {
if (TX_flag == 1) {
CelpBWSenhFrame();
}
if (TX_flag == 2) {
SID_BandScalable_LSP();
}
}
}		

3.3.2.2.3 Syntax elements for non-active frames

Table 3.45 — Syntax of SID_LSP_VQ_ESC0 ()

Syntax	No. of bits	Mnemonic

SID_LSP_VQ_ESC0()
{
if (SampleRateMode == 8kHz) {
SID_NarrowBand_LSP();
} else {
SID_WideBand_LSP();
}
}		

Table 3.46 — Syntax of SID_NarrowBand_LSP()

Syntax	No. of bits	Mnemonic
SID_NarrowBand_LSP() {		
SID_lpc_indices [0];	**4**	**uimsbf**
SID_lpc_indices [1];	**4**	**uimsbf**
SID_lpc_indices [2];	**7**	**uimsbf**
}		

Table 3.47 — Syntax of SID_BandScalable_LSP()

Syntax	No. of bits	Mnemonic
SID_BandScalable_LSP()		
{		
SID_lpc_indices [3];	**4**	**uimsbf**
SID_lpc_indices [4];	**7**	**uimsbf**
SID_lpc_indices [5];	**4**	**uimsbf**
SID_lpc_indices [6];	**6**	**uimsbf**
}		

Table 3.48 — Syntax of SID_WideBand_LSP()

Syntax	No. of bits	Mnemonic
SID_WideBand_LSP()		
{		
SID_lpc_indices [0];	**5**	**uimsbf**
SID_lpc_indices [1];	**5**	**uimsbf**
SID_lpc_indices [2];	**7**	**uimsbf**
SID_lpc_indices [3];	**7**	**uimsbf**
SID_lpc_indices [4];	**4**	**uimsbf**
SID_lpc_indices [5];	**4**	**uimsbf**
}		

Table 3.49 — Syntax of SID_Frame_ESC0 ()

Syntax	No. of bits	Mnemonic
SID_Frame_ESC0()		
{		
SID_rms_index;	**6**	**uimsbf**
}		

3.4 Semantics

This subclause describes the semantics of the syntactic elements. Bitstream elements are shown in **bold**-face and the help variables that appear in the syntax and are needed to extract the bitstream elements are shown in *italics*.

3.4.1 Header semantics

isBaseLayer A one-bit identifier representing whether the corresponding layer is the base layer (1) or an bandwidth scalable or bitrate scalable enhancement layer (0).

isBWSLayer A one-bit identifier representing whether the corresponding layer is the bandwidth scalable enhancement layer (1) or the bitrate scalable enhancement layer (0).

CELP-BRS-id A two-bit identifier representing the order of the bitrate scalable enhancement layers, where the first enhancement layer has the value of '1'. The value of '0' should not be used.

ExcitationMode A one-bit identifier representing whether the Multi-Pulse Excitation tool or the Regular-Pulse Excitation tool is used.

Table 3.50 — Description of ExcitationMode

ExcitationMode	ExcitationID	Description
0	MPE	MPE tool is used
1	RPE	RPE tool is used

SampleRateMode A one-bit identifier representing the sampling rate. Two sampling rates are supported.

Table 3.51 — Description of SampleRateMode

SampleRateMode	SampleRateID	Description
0	8kHz	8 kHz Sampling rate
1	16kHz	16 kHz Sampling rate

FineRateControl A one-bit flag indicating whether fine rate control in very fine steps is enabled or disabled.

Table 3.52 — Description of FineRateControl

FineRateControl	RateControlID	Description
0	OFF	Fine-rate control is disabled
1	ON	Fine-rate control is enabled

FineRate Control enables the bitrate to be decreased with respect to its anchor bitrate. When transmitting the LPC parameters in every frame, the anchor bitrate will be obtained. The lowest bitrate possible bitrate for each configuration can be obtained by transmitting the LPC parameters in 50% of the frames.

SilenceCompression A one bit identifier indicating whether Silence Compression is used or not.

SilenceCompression	SilenceCompressionID	Description
0	SC_OFF	SilenceCompression is disabled
1	SC_ON	SilenceCompression is enabled

RPE_Configuration This is a 3-bit identifier which configures the MPEG-4 CELP coder using the Regular-Pulse Excitation tool. This parameter directly determines the set of allowed bitrates (Table 3.53) and the number of subframes in a CELP frame (Table 3.54).

Table 3.53 — Rate allocation for the 16 kHz mode I coder

RPE_Configuration	Fixed bitrate FineRate Control OFF (bit/s)	Min. bitrate FineRate Control ON, 50% LPC (bit/s)	Max. bitrate FineRate Control ON, 100% LPC (bit/s)
0	14400	13000	14533
1	16000	13900	16200
2	18667	17267	18800
3	22533	21133	22667
4 ... 7	Reserved		

MPE_Configuration This is a 5-bit field that configures the MPEG-4 CELP coder using the Multi-Pulse Excitation tool. This parameter determines the variables *nrof_subframes* and *nrof_subframes_bws*. This parameter also specifies the number of bits for **shape_positions[**i**]**, **shape_signs[**i**]**, **shape_enh_positions[**i**][**j**]** and **shape_enh_signs[**i**][**j**]**.

nrof_subframes is a help parameter, specifying the number of subframes in a CELP frame, and is used to signal how many times the excitation parameters must be read. For the Regular-Pulse Excitation tool operating at the sampling rate of 16 kHz, this variable is dependent on the **RPE_Configuration** as follows:

Table 3.54 — Definition of nrof_subframes for the 16 kHz mode I coder

RPE_Configuration	nrof_subframes
0	6
1	4
2	8
3	10
4 ... 7	Reserved

For the Multi-Pulse Excitation tool, it is derived from the **MPE_Configuration** depending on the sampling rate as follows:

Table 3.55 — Definition of nrof_subframes for the 8 kHz mode II coder

MPE_Configuration	nrof_subframes
0,1,2	4
3,4,5	3
6 ... 12	2
13 ... 21	4
22 ... 26	2
27	4
28 ... 31	reserved

Table 3.56 — Definition of nrof_subframes for the 16 kHz mode II coder

MPE_Configuration	nrof_subframes
0 ... 6	4
8 ... 15	8
16 ... 22	2
24 ... 31	4
7, 23	reserved

NumEnhLayers This is a two-bit field specifying the number of enhancement layers that are used.

Table 3.57 — Definition of nrof_enh_layers

NumEnhLayers	nrof_enh_layers
0	0
1	1
2	2
3	3

BandwidthScalabilityMode This is a one-bit identifier that indicates whether bandwidth scalability is enabled. This mode is only valid when **ExcitationMode** = MPE.

Table 3.58 — Description of BandwidthScalabilityMode

BandwidthScalabilityMode	ScalableID	Description
0	OFF	Bandwidth scalability is disabled

1	ON	Bandwidth scalability is enabled

BWS_Configuration This is a two-bit field that configures the bandwidth extension tool. This identifier is only valid when **BandwidthScalabilityMode** = ON. This parameter specifies the number of bits for **shape_bws_positions[i], shape_bws_signs[i]**.

nrof_subframes_bws This parameter, which is a help variable, represents the number of subframes in the bandwidth extension tool and is derived from the MPE_Configuration as follows:

Table 3.59 — Definition of nrof_subframes_bws

MPE_Configuration	nrof_subframes_bws
0,1,2	8
3,4,5	6
6 ... 12	4
13 ... 21	4
22 ... 26	2
27	not valid
28 ... 31	reserved

3.4.2 Frame semantics

interpolation_flag This is a one-bit flag. When set, it indicates that the LPC parameters for the current frame must be derived using interpolation.

Table 3.60 — Description of interpolation_flag

interpolation_flag	InterpolationID	Description
0	OFF	LPC coefficients of the frame do not have to be interpolated
1	ON	LPC coefficients of the frame must be retrieved by interpolation

LPC_Present This bit indicates whether LPC parameters are attached to the current frame. These LPC parameters are either of the current frame or the next frame.

Table 3.61 — Description of LPC_Present

LPC_Present	LPCID	Description
0	NO	Frame does not carry LPC data
1	YES	Frame carries LPC data

Together, the **interpolation_flag** and the **LPC_Present** flag describe how the LPC parameters are to be derived.

Table 3.62 — LPC decoding process described by interpolation_flag and LPC_Present flag

interpolation_flag	LPC_Present	Description
1	1	LPC Parameters of the current frame must be extracted using interpolation. The current frame carries LPC Parameters belonging to the next frame.
1	0	RESERVED
0	1	LPC Parameters of the current frame are present in the current frame.
0	0	LPC Parameters of the previous frame must be used in the current frame.

lpc_indices[] These are multi-bit fields representing LPC coefficients. These contain information needed to extract the LSP coefficients. The exact extraction procedure is described in the Decoding Process.

shape_delay[subframe**]** This bit field represents the adaptive codebook lag. The decoding of this field depends on **ExcitationMode** and **SampleRateMode**.

Table 3.63 — Number of bits for shape_delay[]

ExcitationMode	SampleRateMode	shape_delay[] (bits)
RPE	16 kHz	8
MPE	8 kHz	8
MPE	16 kHz	9

shape_index[subframe**]** This index contains information needed to extract the fixed codebook contribution from the regular pulse codebook. The number of bits consumed by this field depends on the bitrate (derived from the **RPE_configuration**).

Table 3.64 — Number of bits for shape_index[]

RPE_Configuration	number of bits representing shape_index[]
0	11
1	11
2	12
3	12
4 ... 7	Reserved

gain_indices[0**][**subframe**]** These bit fields specify the adaptive codebook gain in the RPE tool using 6 bits. It is read from the bitstream for every subframe.

gain_indices[1**][**subframe**]** These bit fields specify the fixed codebook gain in the RPE tool. It is read from the bitstream for every subframe. The number of bits read to represent this field depends on the subframe number. For the first subframe this is 5 bits, while for the remaining subframes it is 3 bits.

gain_index[subframe**]** This 6 or 7-bit field represents the gains for the adaptive codebook and the multi-pulse excitation for the 8 kHz or 16 kHz sampling rate, respectively.

gain_enh_index[subframe**]** This 4-bit field represents the gain for the enhancement multi-pulse excitation in the CELP coder at 8 kHz.

gain_bws_index[subframe**]** This 11-bit field represents the gains for the adaptive codebook and two multi-pulse excitation in the bandwidth extension tool.

signal_mode This 2-bit field represents the type of signal. This information is used in the MPE tool. The gain codebooks are switched depending on this information.

Table 3.65 — Description of signal_mode

signal_mode	Description
0	Unvoiced
1,2,3	Voiced

rms_index This parameter indicates the rms level of the frame. This information is only utilized in the MPE tool.

shape_positions[subframe**], shape_signs[**subframe**]** These bit fields represent the pulse positions and the pulse signs for the multi-pulse excitation. The length of the bit field is dependent on **MPE_Configurations**.

Table 3.66 — Definitions of shape_positions[] and shape_signs[] for speech sampled at 8 kHz

MPE_Configuration	shape_positions[] (bits)	shape_signs[] (bits)
0	14	3
1	17	4
2	20	5
3	20	5
4	22	6
5	24	7
6	22	6
7	24	7
8	26	8
9	28	9
10	30	10
11	31	11
12	32	12
13	13	4
14	15	5
15	16	6
16	17	7
17	18	8
18	19	9
19	20	10
20	20	11
21	20	12
22	18	8
23	19	9
24	20	10
25	20	11
26	20	12
27	19	6
28 ... 31	reserved	

Table 3.67 — Definitions of shape_positions[] and shape_signs[] for speech sampled at 16 kHz

MPE_Configuration	shape_positions[] (bits)	shape_signs[] (bits)
0, 16	20	5
1, 17	22	6
2, 18	24	7
3, 19	26	8
4, 20	28	9
5, 21	30	10
6, 22	31	11
7, 23	reserved	reserved
8, 24	11	3
9, 25	13	4
10, 26	15	5
11, 27	16	6
12, 28	17	7
13, 29	18	8
14, 30	19	9
15, 31	20	10

shape_enh_positions[subframe**][], shape_enh_signs[**subframe**][]** These bit fields represent the pulse positions and the pulse signs for the multi-pulse excitation in each enhancement layer. The length of the bit field is dependent on **MPE_Configuration**.

Table 3.68 — Definition of shape_enh_positions[][] and shape_enh_signs[][] for speech sampled at 8 kHz

MPE_Configuration	shape_enh_positions[][] (bits)	shape_enh_signs[][] (bits)
0 ... 12	12	4
13 ... 26	4	2
27	not valid	
28 ... 31	reserved	

Table 3.69 — Definition of shape_enh_positions[][] and shape_enh_signs[][] for speech sampled at 16 kHz

MPE_Configuration	shape_enh_positions[][] (bits)	shape_enh_signs[][] (bits)
0 ... 6, 16 ... 22	12	4
8 ... 15, 24 ... 31	4	2
7, 23	reserved	

shape_bws_delay[subframe**]** This 3-bit field is utilized in decoding the adaptive codebook for the bandwidth extension tool. This value indicates the differential lag from the lag described in **shape_delay[]**.

shape_bws_positions[subframe**], shape_bws_signs[**subframe**]** These fields represent the pulse positions and the pulse signs for the multi-pulse excitation in the bandwidth extension tool. The length of the bit field is dependent on **BWS_Configuration**.

Table 3.70 — Definition of shape_bws_positions[] and shape_bws_signs[]

BWS_Configuration	shape_bws_positions[] (bits)	shape_bws_signs[] (bits)
0	22	6
1	26	8
2	30	10
3	32	12

TX_flag Two-bit field indicating the transmission mode.

Table 3.71 — Definition of TX_flag

TX_flag	Transmission mode
0	Non-active frame. No frame energy or LPC indices are transmitted.
1	Active frame.
2	Non-active frame. Frame energy and LPC indices are transmitted.
3	Non-active frame. Only frame energy is transmitted, no LPC indices are transmitted.

SID_rms_index 6-bit field indicating the energy of the frame.

SID_lpc_indices These are multi-bit fields representing LPC coefficients for the non-active LPC frames (TX_flag = 2). These contain information needed to extract the LSP coefficients. Bitstream semantics for the SID_lpc_indices are shown in Table 3.72.

Table 3.72 — Bitstream semantics for the SID_lpc_indices

Coding mode	Sampling rate [kHz]	Band width scalability	Parameter	Description
I	16	Off	SID_lpc_indices[0]	0-4th LSPs of the 1st stage VQ
			SID_lpc_indices[1]	5-9th LSPs of the 1st stage VQ
			SID_lpc_indices[2]	10-14th LSPs of the 1st stage VQ
			SID_lpc_indices[3]	15-19th LSPs of the 1st stage VQ
			SID_lpc_indices[4]	0-4th LSPs of the 2nd stage VQ
			SID_lpc_indices[5]	5-9th LSPs of the 2nd stage VQ
II	8	On, Off	SID_lpc_indices[0]	0-4th LSPs of the 1st stage VQ
			SID_lpc_indices[1]	5-9th LSPs of the 1st stage VQ
			SID_lpc_indices[2]	0-4th LSPs of the 2nd stage VQ
	16	On	SID_lpc_indices[3]	0-9th LSPs of 1st stage VQ
			SID_lpc_indices[4]	10-19th LSPs of 1st stage VQ
			SID_lpc_indices[5]	0-4th LSPs of 2nd stage VQ
			SID_lpc_indices[6]	5-9th LSPs of 2nd stage VQ
		Off	SID_lpc_indices[0]	0-4th LSPs of 1st stage VQ
			SID_lpc_indices[1]	5-9th LSPs of 1st stage VQ
			SID_lpc_indices[2]	10-14th LSPs of 1st stage VQ
			SID_lpc_indices[3]	15-19th LSPs of 1st stage VQ
			SID_lpc_indices[4]	0-4th LSPs of 2nd stage VQ
			SID_lpc_indices[5]	5-9th LSPs of 2nd stage VQ

3.5 MPEG-4 CELP Decoder tools

This subclause provides a brief description of the functionality, parameter definition and the decoding processes of the tools supported by the MPEG-4 CELP core. The description of each tool comprises three parts and an optional part containing tables used by the tool:

1. Tool description: a short description of the functionality of the tool is given together with its interface.

2. Definitions: the input and output parameters as well as help elements of the tool are described here. Each element is either bold or italic. Bold names indicate that the element is read from the bitstream, italic names

indicate auxiliary elements. If elements are already used by another tool, a reference to the previous definition is given.

3. Decoding process: The decoding process is explained here in detail with the aid of mathematical equations and pseudo-C code.

4. Tables: this optional fourth part contains tables that are used by the tool.

3.5.1 General Introduction to the MPEG-4 CELP decoder tool-set

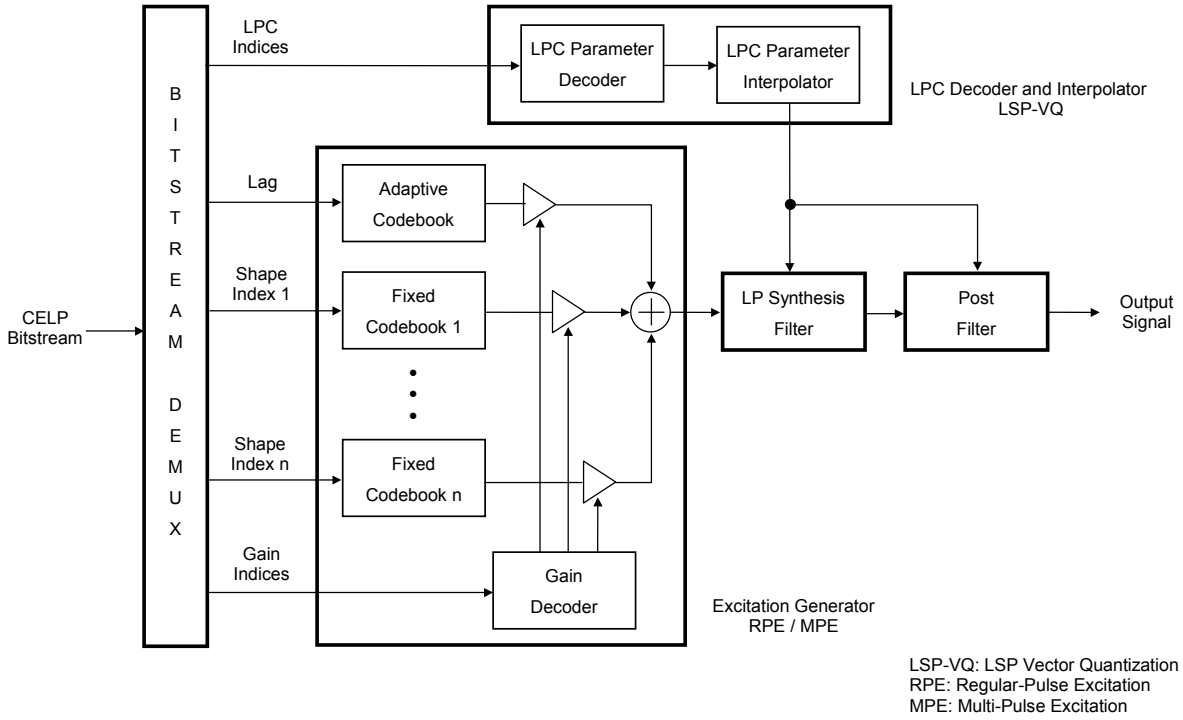


Figure 3.2 — Block diagram illustrating the tools used in the MPEG-4 CELP coder

Figure 3.2 illustrates the MPEG-4 CELP decoder. It operates at a sampling rate of 8 or 16 kHz. One or more individual blocks have been grouped together (outlined in bold), forming the tools available for MPEG-4 CELP decoding. The following tools are supported:

- CELP bitstream demultiplexer
- CELP LPC decoder and interpolator
    - Narrowband LSP-VQ Decoding Tool
    - Wideband LSP-VQ Decoding Tool
    - Bandwidth Scalable LSP-VQ Decoding Tool
- CELP excitation generator
    - Regular-Pulse Excitation Generation Tool
    - Multi-Pulse Excitation Generation Tool
    - Bitrate Scalable Multi-Pulse Excitation Generation Tool
    - Bandwidth Scalable Multi-Pulse Excitation Generation Tool
- CELP LPC synthesis filter
- CELP post-processor (Informative Tool)

The decoding is performed on a frame basis and each frame is divided into subframes. A frame in the bitstream is demultiplexed by the CELP bitstream demultiplexer module. The parameters that are extracted from the bitstream are header information, codes representing LPC coefficients of the frame and the excitation parameters for each subframe. These codes are decoded and interpolated for each subframe by the CELP LPC decoder and interpolator module. For each subframe, the excitation parameters are used to generate the excitation signal using the CELP excitation generator module. The CELP LPC synthesis filter module reconstructs the speech signal on a subframe basis from the interpolated LPC coefficients and the generated excitation signal. Enhancement of the synthesized signal is obtained by the optional CELP post-processor module.

In order to realize Bitrate Scalability, the Bitrate Scalable Multi-Pulse Excitation (MPE) Generation tool is utilized to generate the excitation signal. The Bitrate Scalable MPE Generation tool is realized by adding the enhancement excitation decoding tool to the MPE generation tool in order to enhance the quality of the excitation signal.

The Bandwidth Scalable CELP decoder is realized using both the Bandwidth Scalable LSP-VQ tool and the Bandwidth Scalable MPE Generation tool. These scalable tools are utilized to expand the bandwidth of the decoded signal from 3.4 kHz to 7 kHz.

3.5.2 AAC/CELP scalable configuration

When the Narrowband CELP decoder is used as a “core coder” in the AAC/CELP scalable configuration (see the T/F part) for large-step scalability, the post-filter is switched off. In case the Narrowband CELP decoder operates as a core decoder in a scalable configuration, this decoder may decode the CELP audio object at a sampling rate other than 8 kHz. The sampling frequency of the CELP audio object is specified by **samplingFrequencyIndex** in **AudioSpecificInfo** (See subpart 1). The **SampleRateMode** flag in the CELP header controlling the decoder operation, should indicate 8 kHz. In case the **samplingFrequencyIndex** indicates a sampling rate other than 8000 Hz, the bitrate, the frame length and delay are changed accordingly.

3.5.3 Helping variables

Although each tool has a description of the variables it uses, provided in this subclause are the most commonly used variables shared by multiple tools.

frame_size: This field indicates the number of samples in a frame. The decoder outputs a frame with *frame_size* samples.

nrof_subframes: A frame is built up of a number of subframes. The number of subframes is specified in this field.

sbfrm_size: A subframe consists of a number of samples, which is indicated by this field. The number of samples in a frame must always be equal to the sum of the number of samples in the subframes. Accordingly, the following relation must always hold

frame_size = *nrof_subframes* * *sbfrm_size*

These three parameters depend on the coding mode and the bitrate settings as tabulated in Table 3.73 for Mode I, and Table 3.74 and Table 3.75 for Mode II.

Table 3.73 — CELP coder (Mode I) configuration for 16 kHz sampling rate

RPE_Configuration	Frame_size (#samples)	nrof_subframes	sbfrm_size (#samples)
0	240	6	40
1	160	4	40
2	240	8	30
3	240	10	24
4 ... 7	Reserved		

Table 3.74 — CELP coder (Mode II) configuration for 8 kHz sampling rate

MPE_Configuration	Frame_size (#samples)	nrof_subframes	sbfrm_size (#samples)
0,1,2	320	4	80
3,4,5	240	3	80
6 ... 12	160	2	80
13 ... 21	160	4	40
22 ... 26	80	2	40
27	240	4	60
28 ... 31	Reserved		

Table 3.75 — CELP coder (Mode II) configuration for 16 kHz sampling rate

MPE_Configuration	Frame_size (#samples)	nrof_subframes	sbfrm_size (#samples)
0, ..., 6	320	4	80
8, ..., 15	320	8	40
16, ..., 22	160	2	80
24, ..., 31	160	4	40
7, 23	Reserved		

lpc_order: This field indicates the number of coefficients used for Linear Prediction. The value of this field is 20 for a sampling rate of 16 kHz and 10 for 8 kHz.

num_lpc_indices. This parameter specifies the number of indices containing LPC information that must be read from a bitstream. This is not equal to the LPC-order. The *num_lpc_indices* is 5 in the 8 kHz mode and an additional 6 for the bandwidth extension tool. For a sampling rate of 16 kHz in the Vector Quantizer, this value is 10.

3.5.4 Bitstream elements for the MPEG-4 CELP decoder tool-set

Descriptions of all the bitstream variables are listed in subclause 3.4.

3.5.5 CELP bitstream demultiplexer

3.5.5.1 Tool description

The CELP bitstream demultiplexer tool extracts a CELP frame from the received bitstream.

3.5.5.2 Definitions

All the bitstream elements and the associated help variables have been defined in the subclause 3.4.

3.5.5.3 Decoding process

The decoding of bitstream elements is in accordance with the Syntax described in the subclause 3.3.

3.5.6 CELP LPC decoder and interpolator

The CELP LPC decoder and interpolator has two functions:

1. Retrieves LPC coefficients from the **lpc_indices[]**.

2. Interpolates the retrieved LPC coefficients for every subframe.

Depending on the quantization mode, the **lpc_indices[]** contain information needed to retrieve LSP Coefficients. In the LSP Decoding process, there are three types of decoding processes, namely the Narrowband LSP Decoding process, the Wideband LSP Decoding process and the Bandwidth Scalable LSP Decoding process. The outputs of this process are the filter coefficients, called LPC coefficients, or a-parameters that can be used in the direct-form filter.

Table 3.76 — LPC decoding tools

Coding Mode	Sampling Rate	Tool
Mode I	16 kHz	Wideband LSP-VQ
Mode II	8 kHz 16 kHz 8/16 kHz (BWS)	Narrowband LSP-VQ Wideband LSP-VQ Bandwidth Scalable (BWS) LSP-VQ

3.5.6.1 Narrowband LSP-VQ decoding tool

3.5.6.1.1 Tool description

The Narrowband LSP Decoding tool has a two-stage VQ structure. The quantized LSPs are reproduced by adding LSPs decoded in the first stage and in the second stage. In the second stage there are two decoding processes, with or without interframe prediction, and the decoding process is selected according to the **lpc_indices[4]** flag. Then the decoded LSPs are interpolated and converted to LPC coefficients.

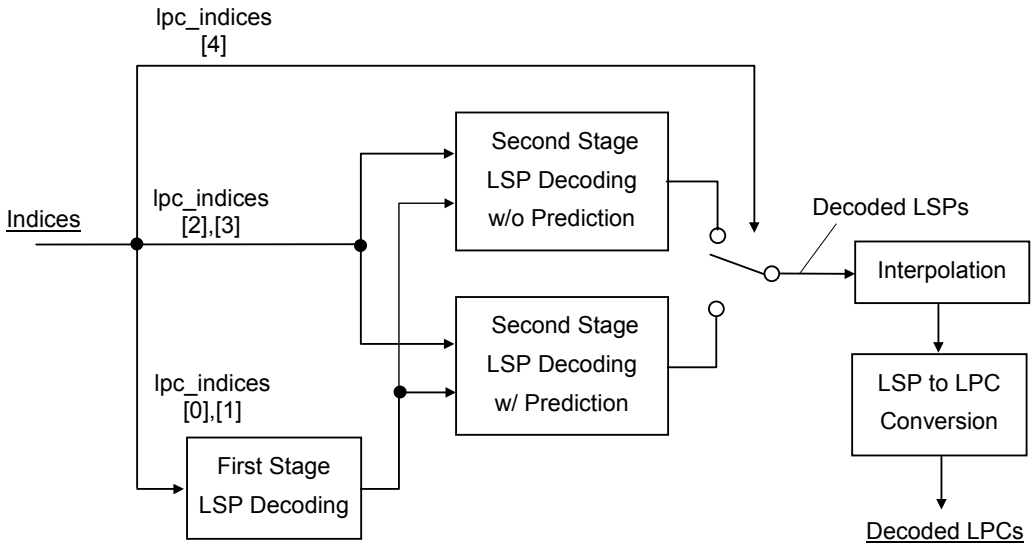


Figure 3.3 — Structure of the narrowband LSP decoder

3.5.6.1.2 Definitions

Input

lpc_indices[]: The dimension of this array is *num_lpc_indices* and contains the packed lpc indices.

Output

int_Qlpc_coefficients[]: This array contains the LPC coefficients for each subframe. The LPC coefficients are decoded and interpolated as described in the decoding process. The LPC coefficients are stacked one after the other in blocks of *lpc_order*. Thus, the dimension of the array is *lpc_order* * *nrof_subframes*.

Configuration

lpc_order: This field indicates the order of LPC being used.

num_lpc_indices: This field contains the number of packed LPC codes. For the Narrowband LSP decoding process, *num_lpc_indices* is set to 5.

nrof_subframes: This field contains the number of subframes.

The following are help elements used in the Narrowband LSP decoding process:

lsp_tbl[][][]	look-up tables for the first stage decoding process
d_tbl[][][]	look-up tables for the second stage decoding process of the VQ without interframe prediction
pd_tbl[][][]	look-up tables for the second stage decoding process of the VQ with interframe prediction.
dim[][]	dimensions for the split vector quantization
sign	sign of code vector for the second stage decoding process
idx	unpacked index for the second stage decoding process
lsp_first[]	the LSPs decoded at the first stage decoding process
lsp_previous[]	the LSPs decoded at the previous frame
lsp_predict[]	the LSPs predicted from *lsp_previous[]* and *lsp_first[]*
lsp_current[]	the LSPs decoded at the current frame
lsp_subframe[][]	the LSPs interpolated at each subframe
ratio_predict	prediction ratio for predicting *lsp_predict[]*
ratio_sub	interpolation ratio for calculating *lsp_subframe[][]*
min_gap	the minimum distance between adjacent LSPs
Convert2lpc()	function for converting LSPs to LPCs

3.5.6.1.3 Decoding process

The LSP decoding process for retrieving interpolated LPC coefficients for each subframe is described below.

3.5.6.1.3.1 Converting indices to LSPs

The LSPs of the current frame (*lsp_current[]*), which are coded by split and two-stage vector quantization, are decoded with a two-stage decoding process. The dimension of each vector is described in Table 3.77 and Table 3.78. The **lpc_indices[0]**,**[1]** and **lpc_indices[2]**,**[3]** represent indices for the first and the second stage respectively.

Table 3.77 — Dimension of the first stage LSP vector

Split Vector Index: i	Vector Dimension: dim[0][i]
0	5
1	5

Table 3.78 — Dimension of the second stage LSP vector

Split Vector Index: i	Vector Dimension: dim[1][i]
0	5
1	5

In the first stage, the LSP vector of the first stage *lsp_first[]* is decoded by looking up the table *lsp_tbl[][][]*. (The table *lsp_tbl[][][]* is shown in Annex 3.C)

```
for(i = 0; i < dim[0][0]; i++)
{
   lsp_first[i] = lsp_tbl[0][lpc_indices[0]][i];
}

for(i = 0; i < dim[0][1]; i++)
{
   lsp_first[dim[0][0]+i] = lsp_tbl[1][lpc_indices[1]][i];
}
```

In the second stage, there are two types of decoding processes, namely, the decoding process of VQ without interframe prediction and of VQ with interframe prediction. The **lpc_indices[4]** indicates which process should be selected.

Table 3.79 — Decoding process for the second stage

LPC Index: lpc_indices[4]	Decoding process
0	VQ without interframe prediction
1	VQ with interframe prediction

Decoding process of VQ without interframe prediction

In order to obtain LSPs of the current frame *lsp_current[]* , the decoded vectors in the second stage are added to the decoded first stage LSP vector *lsp_first[]*. The MSB of the **lpc_indices[]** represents the sign of the decoded vector, and the remaining bits represent the index for the table *d_tbl[][][].* (The table *d_tbl[][][]* is shown in Annex 3.C)

```
sign = lpc_indices[2] >> 6;
idx = lpc_indices[2] & 0x3f;
if (sign==0)
{
   for (i = 0; i < dim[1][0]; i++)
   {
```

```
      lsp_current[i] = lsp_first[i] + d_tbl[0][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current[i] = lsp_first[i] - d_tbl[0][idx][i];
   }
}
sign = lpc_indices[3] >> 5;
idx = lpc_indices[3] & 0x1f;
if (sign==0)
{
   for(i = 0; i < dim[1][1]; i++)
   {
      lsp_current[dim[1][0]+i] = lsp_first[dim[1][0]+i] + d_tbl[1][idx][i];
   }
}
else
{
   for(i = 0; i < dim[1][1]; i++)
   {
      lsp_current[dim[1][0]+i] = lsp_first[dim[1][0]+i] - d_tbl[1][idx][i];
   }
}
```

Decoding process of VQ with interframe prediction

In order to obtain LSPs of the current frame *lsp_current[]* , the decoded vectors of the second stage are added to the LSP vector *lsp_predict[],* which are predicted from the decoded LSPs of the previous frame *lsp_previous[]* and the decoded first stage LSP vector *lsp_first[]*. In the same way as the decoding process of VQ without interframe prediction, the MSB of the **lpc_indices[]** represents the sign of the decoded vector, and the remaining bits represent the index for the table *pd_tbl[][][].* (The table *pd_tbl[][][]* is shown in Annex 3.C)

```
for (i = 0; i < lpc_order; i++)
{
   lsp_predict[i]=(1-ratio_predict)*lsp_first[i]
                  +ratio_predict*lsp_previous[i]
}
```

where *ratio_predict* = 0.5

```
sign = lpc_indices[2] >> 6;
idx = lpc_indices[2] & 0x3f;
if (sign==0)
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current[i] = lsp_predict[i] + pd_tbl[0][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current[i] = lsp_predict[i] - pd_tbl[0][idx][i];
   }
```

```
}
sign = lpc_indices[3] >> 5;
idx = lpc_indices[3] & 0x1f;
if(sign==0)
{
   for(i = 0; i < dim[1][1]; i++)
   {
      lsp_current[dim[1][0]+i] = lsp_predict[dim[1][0]+i] + pd_tbl[1][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][1]; i++)
   {
      lsp_current[dim[1][0]+i] = lsp_predict[dim[1][0]+i] - pd_tbl[1][idx][i];
   }
}
```

3.5.6.1.3.2 Stabilization of LSPs

The decoded LSPs *lsp_current[]* are stabilized in order to ensure stability of the LPC synthesis filter, which is derived from the decoded LSPs. The decoded LSPs are arranged in ascending order, having a distance of at least *min_gap* between adjacent coefficients.

```
for(i = 0; i < lpc_order; i++)
{
   if (lsp_current[i] < min_gap)
   {
      lsp_current[i] = min_gap;
   }
}
for (i = 0; i < lpc_order-1; i++)
{
   if (lsp_current[i+1]-lsp_current[i] < min_gap)
   {
      lsp_current[i+1] = lsp_current[i]+min_gap;
   }
}
for (i = 0; i < lpc_order; i++)
{
   if (lsp_current[i] > 1-min_gap)
   {
      lsp_current[i] = 1-min_gap;
   }
}
for (i = lpc_order-1; i > 0; i--)
{
   if (lsp_current[i]-lsp_current[i-1] < min_gap)
   {
      lsp_current[i-1] = lsp_current[i]-min_gap;
   }
}
```

where min_gap = 2.0/256.0

3.5.6.1.3.3 Interpolation of LSPs

The decoded LSPs *lsp_current[]* are interpolated linearly at each subframe using the LSPs of the previous frame *lsp_previous[]*.

```
for (n = 0; n < nrof_subframes; n++)
{
   ratio_sub=(n+1)/nrof_subframes
   for(i = 0; i < lpc_order; i++)
   {
      lsp_subframe[n][i] = ((1-ratio_sub)*lsp_previous[i]+ratio_sub*lsp_current[i]));
   }
}
```

3.5.6.1.3.4 LSP to LPC conversion

The interpolated LSPs are converted to the LPCs using the auxiliary function *Convert2lpc().*

```
for (n = 0; n < nrof_subframes; n++)
{
   Convert2lpc (lpc_order, lsp_subframe[n], int_Qlpc_coefficients + n*lpc_order]);
}
```

The LSP to LPC conversion is described below.The input LSPs must be normalized in the range of zero to π.

```
tmp_lpc[0] = 1.0;
for (i = 1; i < lpc_order + 1; i++) tmp_lpc[i] = 0.0;
for (i = 0; i < (lpc_order+1)*2; i++) w[i] = 0.0;

for (j = 0; j < lpc_order + 1; j++)
{
   xin1 = tmp_lpc[j];
   xin2 = tmp_lpc[j];
   for (i = 0; i < (lpc_order >> 1); i++)
   {
      n1 = i*4;
      n2 = n1+1;
      n3 = n2+1;
      n4 = n3+1;
      xout1 = -2. * cos(lsp_coefficients[i*2+0]) * w[n1] + w[n2] + xin1;
      xout2 = -2. * cos(lsp_coefficients[i*2+1]) * w[n3] + w[n4] + xin2;
      w[n2] = w[n1];
      w[n1] = xin1;
      w[n4] = w[n3];
      w[n3] = xin2;
      xin1 = xout1;
      xin2 = xout2;
   }

   xout1 = xin1 + w[n4+1];
   xout2 = xin2 - w[n4+2];
   tmp_lpc[j] = 0.5 * (xout1 + xout2);

   w[n4+1] = xin1;
   w[n4+2] = xin2;
}
```

```
for (i = 0; i < lpc_order; i++)
{
   lpc_coefficients[i] = tmp_lpc[i+1];
}
```

3.5.6.1.3.5 Storing the coefficients

After calculation of the LPC coefficients, the current LSPs must be stored in memory, since they are used for interpolation at the next frame.

```
for (i = 0; i < lpc_order; i++)
{
   lsp_previous[i] = lsp_current[i];
}
```

It should be noted that the stored LSPs *lsp_previous[]* must be initialized as described below when the entire decoder is initialized.

```
for (i = 0; i < lpc_order; i++)
{
   lsp_previous[i] = (i+1) / (lpc_order+1);
}
```

3.5.6.2 Wideband LSP-VQ decoding tool

3.5.6.2.1 Tool description

The Wideband LSP decoding process is based on the Narrowband LSP Decoding process. As described in Figure 3.4, the Wideband LSP Decoding process consists of two LSP decoding blocks connected in parallel. Each of the decoding blocks is identical to the Narrowband LSP Decoding process and decodes the lower and the upper part of the LSPs respectively. The decoded LSPs are combined and output as one set of the parameters.

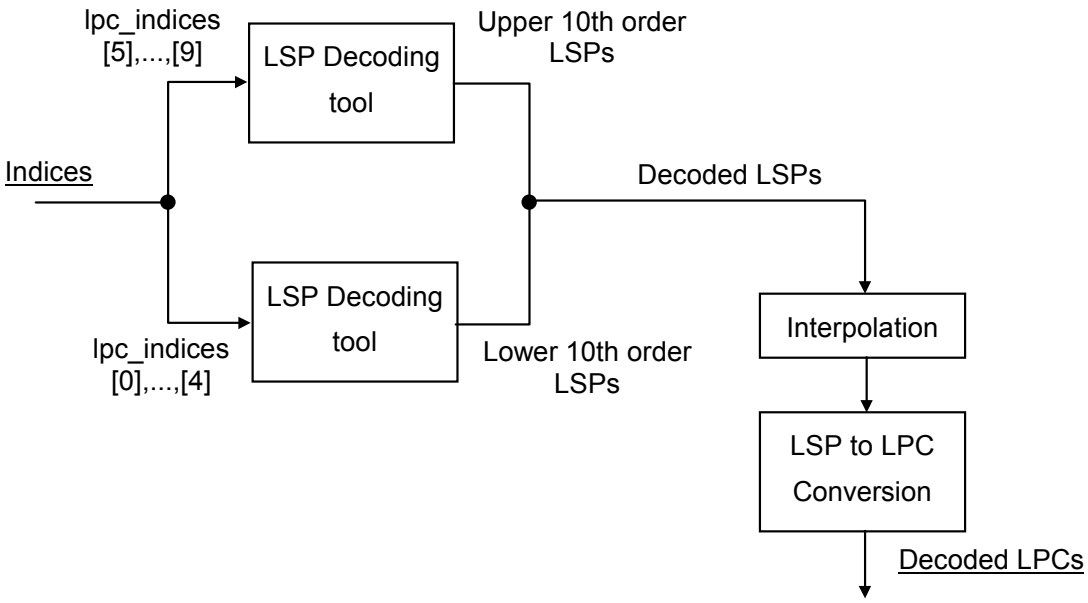


Figure 3.4 — Structure of the wideband LSP decoder

3.5.6.2.2 Definitions

Input

lpc_indices[]: The dimension of this array is *num_lpc_indices* and contains the packed lpc indices.

Output

int_Qlpc_coefficients[]: This array contains the LPC coefficients for each subframe. The LPC coefficients are decoded and interpolated as described in the decoding process. The LPC coefficients are stacked one after the other in blocks of *lpc_order*. Thus, the dimension of the array is *lpc_order * nrof_subframes*.

Configuration

lpc_order: This field indicates the order of LPC being used.

num_lpc_indices: This field contains the number of packed LPC codes. For the Wideband LSP Decoding process, *num_lpc_indices* is set to 10.

nrof_subframes: This field contains the number of subframes.

The following are help elements used in the Wideband LSP Decoding process:

lsp_tbl[][][]	look-up tables for the first stage decoding process
d_tbl[][][]	look-up tables for the second stage decoding process of the VQ without interframe prediction
pd_tbl[][][]	look-up tables for the second stage decoding process of the VQ with interframe prediction.
dim[][]	dimensions for the split vector quantization

sign	sign of code vector for the second stage decoding process
idx	unpacked index for the second stage decoding process
lsp_first[]	the LSPs decoded at the first stage decoding process
lsp_previous[]	the LSPs decoded at the previous frame
lsp_predict[]	the LSPs predicted from *lsp_previous[]* and *lsp_first[]*
lsp_current[]	the LSPs decoded at the current frame
lsp_current_lower[]	the lower part of the current LSPs
lsp_current_upper[]	the upper part of the current LSPs
lsp_subframe[][]	the LSPs interpolated at each subframe
ratio_predict	prediction ratio for predicting *lsp_predict[]*
ratio_sub	interpolation ratio for calculating *lsp_subframe[][]*
min_gap	the minimum distance between adjacent LSPs
Convert2lpc()	function for converting LSPs to LPCs

3.5.6.2.3 Decoding process

The LSP decoding process for retrieving interpolated LPC coefficients for each subframe is described below.

3.5.6.2.3.1 Converting indices to LSPs

Using the same manner as the Narrowband LSP Decoding process, The LSPs of the current frame (*lsp_current[]*), which are coded by split and two-stage vector quantization, are decoded with a two-stage decoding process.

Firstly, the lower part of the current LSPs *lsp_current_lower[]* are decoded. The dimension of each vector is described in Table 3.80 and Table 3.81. The **lpc_indices[0]**,**[1]** and **lpc_indices[2]**,**[3]** represent indices for the first and the second stage respectively.

Table 3.80 — Dimension of the first stage LSP vector

Split Vector Index: i	Vector Dimension: dim[0][i]
0	5
1	5

Table 3.81 — Dimension of the second stage LSP vector

Split Vector Index: i	Vector Dimension: dim[1][i]
0	5
1	5

In the first stage, the LSP vector of the first stage *lsp_first[]* is decoded by looking up the table *lsp_tbl[][][]*. (The table *lsp_tbl[][][]* is shown in Annex 3.C)

```
for (i = 0; i < dim[0][0]; i++)
{
   lsp_first[i] = lsp_tbl[0][lpc_indices[0]][i];
}

for (i = 0; i < dim[0][1]; i++)
{
   lsp_first[dim[0][0]+i] = lsp_tbl[1][lpc_indices[1]][i];
}
```

In the second stage, the **lpc_indices[4]** indicates which process should be selected.

Table 3.82 — Decoding process for the second stage

LPC Index: lpc_indices[4]	Decoding process
0	VQ without interframe prediction
1	VQ with interframe prediction

Decoding process of VQ without interframe prediction

In order to obtain LSPs of the current frame *lsp_current_lower[]* , the decoded vectors in the second stage are added to the decoded first stage LSP vector *lsp_first_lower[]*. The MSB of the **lpc_indices[]** represents the sign of the decoded vector, and the remaining bits represent the index for the table *d_tbl[][][].* (The table *d_tbl[][][]* is shown in Annex 3.C)

```
sign = lpc_indices[2] >> 6;
idx = lpc_indices[2] & 0x3f;
if (sign == 0)
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current_lower[i] = lsp_first[i] + d_tbl[0][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current_lower[i] = lsp_first[i] - d_tbl[0][idx][i];
   }
}
sign = lpc_indices[3] >> 6;
idx = lpc_indices[3] & 0x3f;
if (sign == 0)
{
   for (i = 0; i < dim[1][1]; i++)
   {
      lsp_current_lower[dim[1][0]+i] = lsp_first[dim[1][0]+i] + d_tbl[1][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][1]; i++)
   {
      lsp_current_lower[dim[1][0]+i] = lsp_first[dim[1][0]+i] - d_tbl[1][idx][i];
   }
}
```

Decoding process of VQ with interframe prediction

In order to obtain LSPs of the current frame *lsp_current_lower[]* , the decoded vectors of the second stage are added to the LSP vector *lsp_predict[],* which are predicted from the decoded LSPs of the previous frame *lsp_previous[]* and the decoded first stage LSP vector *lsp_first[]*. In the same way as the decoding process of VQ without interframe prediction, the MSB of the **lpc_indices[]** represents the sign of the decoded vector, and the remaining bits represent the index for the table *pd_tbl[][][].* (The table *pd_tbl[][][]* is shown in Annex 3.C)

```
for (i = 0; i < lpc_order/2; i++)
{
   lsp_predict[i]=(1-ratio_predict)*lsp_first[i] + ratio_predict*lsp_previous[i];
}
```

where *ratio_predict* = 0.5

```
sign = lpc_indices[2] >> 6;
idx = lpc_indices[2] & 0x3f;
if (sign == 0)
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current_lower[i] = lsp_predict[i] + pd_tbl[0][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current_lower[i] = lsp_predict[i] - pd_tbl[0][idx][i];
   }
}
sign = lpc_indices[3] >> 6;
idx = lpc_indices[3] & 0x3f;
if (sign == 0)
{
   for (i = 0; i < dim[1][1]; i++)
   {
      lsp_current_lower[dim[1][0]+i] = lsp_predict[dim[1][0]+i] + pd_tbl[1][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][1]; i++)
   {
      lsp_current_lower[dim[1][0]+i] = lsp_predict[dim[1][0]+i] - pd_tbl[1][idx][i];
   }
}
```

Next, the upper part of the current LSPs are decoded in the same manner as the decoding process of the lower part. The dimension of each vector is described in Table 3.83 and Table 3.84. The **lpc_indices[5]**,**[6]** and **lpc_indices[7]**,**[8]** represent indices for the first and the second stage respectively.

Table 3.83 — Dimension of the first stage LSP vector

Split Vector Index: i	Vector Dimension: dim[0][i]

0	5
1	5

Table 3.84 — Dimension of the second stage LSP vector

Split Vector Index: i	Vector Dimension: dim[1][i]
0	5
1	5

```
for (i = 0; i < dim[0][0]; i++)
{
   lsp_first[i] = lsp_tbl[0][lpc_indices[5]][i];
}

for (i = 0; i < dim[0][1]; i++)
{
   lsp_first[dim[0][0]+i] = lsp_tbl[1][lpc_indices[6]][i];
}
```

In the second stage, the **lpc_indices[9]** indicates which process should be selected.

Table 3.85 — Decoding process for the second stage

LPC Index: lpc_indices[9]	Decoding process
0	VQ without interframe prediction
1	VQ with interframe prediction

Decoding process of VQ without interframe prediction

```
sign = lpc_indices[7] >> 6;
idx = lpc_indices[7] & 0x3f;
if (sign == 0)
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current_upper[i] = lsp_first[i] + d_tbl[0][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current_upper[i] = lsp_first[i] - d_tbl[0][idx][i];
   }
}
sign = lpc_indices[8] >> 4;
idx = lpc_indices[8] & 0x0f;
if (sign == 0)
{
   for (i = 0; i < dim[1][1]; i++)
   {
      lsp_current_upper[dim[1][0]+i] = lsp_first[dim[1][0]+i] + d_tbl[1][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][1]; i++)
   {
      lsp_current_upper[dim[1][0]+i] = lsp_first[dim[1][0]+i] - d_tbl[1][idx][i];
   }
}
```

Decoding process of VQ with interframe prediction

```
for (i = 0; i < lpc_order/2; i++)
{
   lsp_predict[i]=(1-ratio_predict)*lsp_first[i] +ratio_predict*lsp_previous[lpc_order/2+i];
}
```

where *ratio_predict* = 0.5

```
sign = lpc_indices[7] >> 6;
idx = lpc_indices[7] & 0x3f;
if (sign == 0)
{
   for (i = 0; i < dim[1][0]; i++)
   {
      lsp_current_upper[i] = lsp_predict[i] + pd_tbl[0][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][0]; i++)
   {
```

```
      lsp_current_upper[i] = lsp_predict[i] - pd_tbl[0][idx][i];
   }
}
sign = lpc_indices[8] >> 4;
idx = lpc_indices[8] & 0x0f;
if (sign == 0)
{
   for (i = 0; i < dim[1][1]; i++)
   {
      lsp_current_upper[dim[1][0]+i] = lsp_predict[dim[1][0]+i] + pd_tbl[1][idx][i];
   }
}
else
{
   for (i = 0; i < dim[1][1]; i++)
   {
      lsp_current_upper[dim[1][0]+i] = lsp_predict[dim[1][0]+i] - pd_tbl[1][idx][i];
   }
}
```

Finally, the decoded LSPs *lsp_current_lower[]* and *lsp_current_upper[]* are combined and stored to the array *lsp_current[].*

```
for (i = 0; i < lpc_order/2; i++)
{
   lsp_current[i] = lsp_current_lower[i];
}
for (i = 0; i < lpc_order/2; i++)
{
   lsp_current[lpc_order/2 + i] = lsp_current_upper[i];
}
```

3.5.6.2.4 Stabilization of LSPs

The decoded LSPs *lsp_current[]* are stabilized in order to ensure stability of the LPC synthesis filter, which is derived from the decoded LSPs. The decoded LSPs are arranged in ascending order, having a distance of at least *min_gap* between adjacent coefficients.

```
for (i = 0; i < lpc_order; i++)
{
   if (lsp_current[i] < min_gap)
   {
      lsp_current[i] = min_gap;
   }
}

for (i = 0; i < lpc_order - 1; i++)
{
   if (lsp_current[i+1]-lsp_current[i] < min_gap)
   {
      lsp_current[i+1] = lsp_current[i]+min_gap;
   }
}
```

```
for (i = 0; i < lpc_order; i++)
{
   if (lsp_current[i] > 1-min_gap)
   {
      lsp_current[i] = 1-min_gap;
   }
}

for (i = lpc_order - 1; i > 0; i--)
{
   if (lsp_current[i]-lsp_current[i-1] < min_gap)
   {
      lsp_current[i-1] = lsp_current[i]-min_gap;
   }
}
```

where lpc_order = 20 and min_gap = 1.0/256.0

3.5.6.2.5 Interpolation of LSPs

The decoded LSPs *lsp_current[]* are interpolated linearly at each subframe using the LSPs of the previous frame *lsp_previous[]*.

```
for (n = 0; n < nrof_subframes; n++)
{
   ratio_sub=(n+1)/nrof_subframes
   for (i = 0; i < lpc_order; i++)
   {
      lsp_subframe[n][i] = ((1-ratio_sub)*lsp_previous[i]+ratio_sub*lsp_current[i]));
   }
}
```

3.5.6.2.6 LSP to LPC conversion

The interpolated LSPs are converted to the LPCs using the auxiliary function *Convert2lpc().*

```
for (n = 0; n < nrof_subframes; n++)
{
   Convert2lpc (lpc_order, lsp_subframe[n], int_Qlpc_coefficients + n*lpc_order]);
}
```

3.5.6.2.7 Storing the coefficients

After calculation of the LPC coefficients, the current LSPs must be stored in memory, since they are used for interpolation at the next frame.

```
for (i = 0; i < lpc_order; i++)
{
   lsp_previous[i] = lsp_current[i];
}
```

It should be noted that the stored LSPs *lsp_previous[]* must be initialized as described below when the entire decoder is initialized.

```
for (i = 0; i < lpc_order; i++)
{
   lsp_previous[i] = (i+1) / (lpc_order+1);
}
```

3.5.6.3 Bandwidth scalable LSP-VQ decoding tool

3.5.6.3.1 Tool description

The Bandwidth Scalable LSP-VQ decoding tool is utilized to add bandwidth scalability to the Mode II coder. In the Bandwidth Scalable LSP-VQ decoding tool, the bandwidth extension tool is connected with the Narrowband LSP-VQ decoding tool as shown in Figure 3.5. The LPC coefficients for each subframes are reconstructed by retrieving **lpc_indices[]** and converting the decoded LSPs in the narrowband mode.

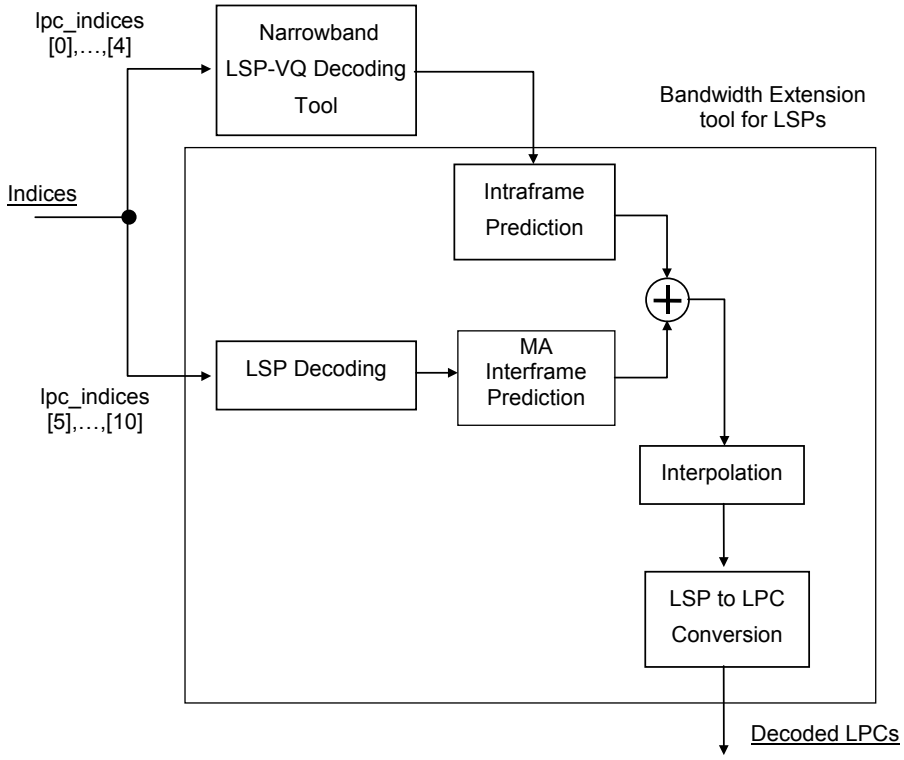


Figure 3.5 — Blockdiagram of the bandwidth scalable LSP-VQ decoding tool

3.5.6.3.2 Definitions

Input

lpc_indices[]: The dimension of this array is *num_lpc_indices* and contains the packed lpc indices.

lsp_current[]: This array contains the decoded LSP parameters, which are normalized in the range of zero to PI, at the Narrowband LSP Decoding tool. These parameters are obtained as intermediate parameters in the Narrowband LSP Decoding process described in subclause 3.5.6.1 and forwarded to the Bandwidth Scalable LSP Decoding tool.

Output

int_QIpc_coefficients[]: This array contains the LPC coefficients for each subframe. The LPC coefficients are decoded and interpolated as described in the decoding process. The LPC coefficients are stacked one after the other in blocks of *lpc_order*. Thus, the dimension of the array is *lpc_order * nrof_subframes*.

Configuration

lpc_order: This field indicates the order of LPC being used.

num_lpc_indices: This field contains the number of packed LPC codes. For the Bandwidth Scalable LSP Decoding process, *num_lpc_indices* is set to 11.

nrof_subframes: This field contains the number of subframes.

nrof_subframes_bws: This parameter, which is a help variable, represents the number of subframes in the bandwidth extension tool.

The following help elements are used in the decoding procedure.

lsp_bws_tbl[][][] look-up tables for bandwidth scalable LSP decoding process

lsp_bws_buf[][]	buffer containing LSP prediction residual
bws_ma_prdct[][]	prediction coefficients for moving average prediction
bws_nw_prdct[]	prediction coefficients for conversion for LSPs from narrowband to wideband
lsp_current[]	the decoded LSPs in the narrowband CELP coder
lsp_bws_current[]	the decoded LSPs at the current frame in the bandwidth extension tool
lsp_bws_previous[]	the decoded LSPs at the previous frame in the bandwidth extension tool
lpc_order_bws	the order of LPC in the bandwidth extension tool (=20). This is twice the *lpc_order* in narrowband CELP.

3.5.6.3.3 Decoding process

The LPC coefficients at each subframe are reproduced using **lpc_indices[5], ... , lpc_indices[10]** and the current LSPs (*lsp_current[]*) for the narrowband CELP coder. The decoding procedure consists of three steps, decoding LSPs at last subframe, interpolation for the other subframes and conversion of LSPs to LPC coefficients.

Decoding process of the bandwidth extension tool for LSP parameters

The intraframe prediction module produces estimated LSP parameters by converting the decoded LSP parameters (*lsp_current[]*) at the Narrowband LSP-VQ decoding tool. The estimated residual LSP parameters are decoded from **lpc_indices[]** by means of the moving average (MA) interframe prediction.

The LSPs at the last subframe (*lsp_bws_current[]*) for the bandwidth extension tool are reconstructed by table look up and prediction as follows:

```
for (i = 0; i < 5; i++)
{
   lsp_bws_buf[0][i] = lsp_bws_tbl[0][lpc_indices[5]][i]+lsp_bws_tbl[2][lpc_indices[7]][i];
}

for (i = 5; i < 10; i++)
{
   lsp_bws_buf[0][i] = lsp_bws_tbl[0][lpc_indices[5]][i]+lsp_bws_tbl[3][lpc_indices[8]][i-5];
}

for (i = 10; i < 15; i++)
{
   lsp_bws_buf[0][i] = lsp_bws_tbl[1][lpc_indices[6]][i-10]+lsp_bws_tbl[4][lpc_indices[9]][i-10];
}

for (i = 15; i < 20; i++)
{
   lsp_bws_buf[0][i] = lsp_bws_tbl[1][lpc_indices[6]][i-10]+lsp_bws_tbl[5][lpc_indices[10]][i-15];
}

for (n = 0; n <= 2; n++)
{
   for (i = 0; i < 20; i++)
   {
      lsp_bws_current[i] += bws_ma_prdct[n][i]*lsp_bws_buf[n][i];
   }
}

for (i = 0; i < 10; i++)
{
   lsp_bws_current[i] += bws_nw_prdct[i]*lsp_current[i];
}
```

where *lsp_bws_tbl[][][]* are LSP codebooks listed in Annex 3.C. *bws_ma_prdct[][]* and *bws_nw_prdct[]* are prediction coefficients for moving average interframe prediction and intraframe prediction, respectively. *lsp_bws_buf[][]* is a buffer containing LSP prediction residual at the current frame and previous two ones. This buffer is shifted for the next frame operation as follows:

```
for (n = 2; n > 0; n--)
{
   for (i = 0; i < 20; i++)
   {
      lsp_bws_buf[n][i] = lsp_bws_buf[n-1][i];
   }
}
```

The decoded LSPs *lsp_bws_current[]* are stabilized in order to ensure stability of the LPC synthesis filter, which is derived from the decoded LSPs. The decoded LSPs are arranged in ascending order, having a distance of at least *min_gap* between adjacent coefficients.

```
for (i = 0; i < lpc_order_bws; i++)
{
   if (lsp_bws_current[i] < 0.0 || lsp_bws_current[i] > PAI)
   {
```

```
        lsp_bws_current[i] = 0.05 + PAI * i / lpc_order_bws;
   }
}

for (i = (lpc_order_bws-1); i > 0; i--)
{
   for (j = 0; j < i; j++)
   {
      if (lsp_bws_current[j] + min_gap > lsp_bws_current[j+1])
      {
         tmp = 0.5 * (lsp_bws_current[j] + lsp_bws_current[j+1]);
         lsp_bws_current[j] = tmp - 0.51 * min_gap;
         lsp_bws_current[j+1] = tmp + 0.51 * min_gap;
      }
   }
}
```

where PAI = 3.141592 and min_gap = 0.028.

Interpolation of the LSP parameters

The decoded LSPs are interpolated linearly at each subframe.

```
for (n = 0; n < nrof_subframes_bws; n++)
{
   ratio_sub = (n+1)/nrof_subframes_bws;
   for (i = 0; i < 2*lpc_order; i++)
   {
      lsp_bws_subframe[n][i]=((1-ratio_sub)*lsp_bws_previous[i]
                              + ratio_sub*lsp_bws_current[i]));
   }
}

for (i = 0; i < 2*lpc_order; i++)
{
   lsp_bws_previous[i] = lsp_bws_subframe[nrof_subframes_bws-1][i];
}
```

Conversion of LSPs to LPC coefficients

The interpolated LSPs are converted to the LPC coefficients at each subframes.

```
for (n = 0; n < nrof_subframes_bws; n++)
{
   Convert2lpc (lpc_order_bws,lsp_bws_subframe[n],
                            &int_Qlpc_coefficients[n*lpc_order_bws]);
}
```

3.5.6.4 Fine rate control in the LSP decoding tool

3.5.6.4.1 Tool description

Fine Rate Control is available with the LSP Decoding tool. If the FRC is used, two additional parameters, **interpolation_flag** and **LPC_present,** are included in the input parameters. The Narrowband LSP Decoding process is carried out if the **lpc_indices[]** are present in the current frame (**LPC_present** = YES).

3.5.6.4.2 Definitions

Input

lpc_indices[]: The dimension of this array is *num_lpc_indices* and contains the LPC indices.

interpolation_flag: This is a one-bit flag. When set, the flag indicates that the frame under consideration is an incomplete frame, i.e. the frame does not carry the LPC coefficients of the current speech frame, but only its excitation parameters (adaptive and fixed codebook parameters). The LPC coefficients for the speech frame under consideration should be obtained using interpolation of the LPC coefficients of the adjacent frames.

1 LPC coefficients of the frame must be retrieved by interpolation

0 LPC coefficients of the frame do not have to be interpolated.

For maintaining good subjective quality, there may never be more than one frame in succession without the LPC information, i.e. the **interpolation_flag** may not have the value 1 in two successive CelpFrames.

LPC_Present This field indicates the presence of LPC parameters in the speech frame under consideration. These LPC coefficients are either of the current speech frame or those of a subsequent frame. When used in combination with the **interpolation_flag,** the two parameters completely describe how the LPC coefficients of the current frame are derived. If the interpolation flag is set, the LPC coefficients of the current frame are calculated by using the LPC coefficients of the previous and next frame. Generally, this would mean that the decoding of the current frame must be delayed by one frame. To avoid this additional delay in the decoder, the LPC coefficients of the next frame are enclosed in the current frame. In this case, the **LPC_Present** flag is set. Since the LPC coefficients of the next frame are already present in the current frame, the next frame will not contain LPC information.

- If the **interpolation_flag** is “1” and the **LPC_Present** is “1”, then (a) the LPC parameters of the current frame are derived using the LPC parameters of the previous frame and that of the next frame and (b) the current frame (frame under consideration) carries LPC parameters of a subsequent frame, but not those of the frame under consideration. The LPC coefficients of the frame under consideration are obtained by interpolation of the previously received LPC parameters and the LPC parameters received in the frame under consideration.

- If the **interpolation_flag** is “0” and the **LPC_Present** is “0”, the LPC parameters to be used with the frame under consideration are those received in the previous frame.

- When **interpolation_flag** is “0” and the **LPC_Present** is “1”, then the current frame is a complete frame and the LPC parameters received in the current frame belong to the current frame.

Such a construction is chosen so as to minimize the delay when the decoder begins reconstructing the frame, the LPC coefficients of which are obtained using interpolation without having to wait for the next frame to arrive. Secondly, such a combination makes it possible to decode the bitstream from any point (random access). In the fixed bitrate configuration, these two flags exhibit a fixed pattern

01, 01, 01, 01, 01, 01, … The string 01 is repeated

11, 00, 11, 00, 11, 00, … The string 11, 00 is repeated (Fixed bitrate achieved over two frames)

For variable bitrate (when **FineRateControl** = ON), the string will, generally, not exhibit a fixed pattern.

Output

int_Qlpc_coefficients[]: This array contains the LPC coefficients for each subframe. The LPC coefficients are decoded and interpolated as described in the decoding process.

3.5.6.4.3 Decoding process

If the **interpolation_flag** is set, the LSPs decoded in the current frame belong to the next frame and LSPs for the current frame are calculated by linearly interpolating the LSPs of the adjacent frames.

If the **interpolation_flag** is not set and the **lpc_indices[]** are present in the current frame, the computed coefficients belong to the current frame and no interpolation need be performed.

If the **interpolation_flag** is not set and the **lpc_indices[]** are not present in the current frame, the LSPs for the current frame are already received in the previous frame. Therefore the LSPs received in the previous frame are copied and used in the current frame.

3.5.7 CELP excitation generator

The CELP excitation generator generates the excitation signal for one subframe from the received indices using either regular pulse excitation (RPE) process or a multi-pulse excitation (MPE) process depending on the coding mode.

Table 3.86 — Excitation generation tools

Coding Mode	Sampling Rate	Tool
Mode I	16 kHz	RPE
Mode II	8, 16 kHz 8, 16 kHz 8/16 kHz (BWS)	MPE Bitrate Scalable MPE Bandwidth Scalable MPE

3.5.7.1 Regular pulse excitation generation tool

3.5.7.1.1 Tool description

Figure 3.6 illustrates the excitation generation process used for the 16 kHz sampling rate. The excitation signal is constructed from a periodic component (adaptive codebook contribution) and non periodic component (RPE contribution) scaled by their respective gains. Using the **shape_delay[***sub_frame***]** and the **gain_indices[0][***sub_frame***]**, the adaptive codebook contribution is computed. The RPE contribution is computed by using **shape_index[***sub_frame***]** and **gain_indices[1][***sub_frame***]**. For clarity, the indexing based on *sub_frame* is dropped. It is noted that the excitation generation process is repeated every subframe.

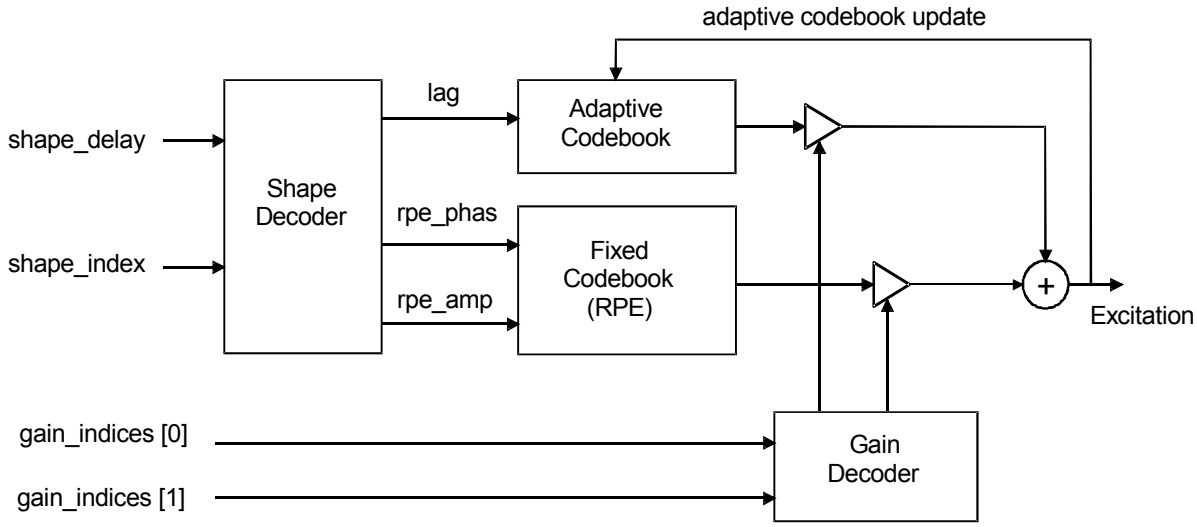


Figure 3.6 — Regular pulse excitation generator

3.5.7.1.2 Definitions

Input

shape_delay[]: This array is of dimension *nrof_subframes* and contains the adaptive codebook lag.

shape_index[]: This array is of dimension *nrof_subframes* and contains the RPE codebook index.

gain_indices[0][]: This array is of dimension *nrof_subframes* and contains the adaptive codebook gain index.

gain_indices[1][]: This array is of dimension *nrof_subframes* and contains the RPE codebook gain.

Output

excitation[]: This array is of dimension *sbfrm_size* and contains the excitation signal. This signal is reconstructed from shape and gain vectors using the adaptive and fixed codebooks.

lag: This field contains the decoded lag (pitch period) for adaptive codevector.

adaptive_gain: This field contains the decoded gain for adaptive codevector.

Configuration

sbfrm_size: This field indicates the number of samples in the subframe.

nrof_subframes: This field indicates the number of subframes.

The additional elements used in the RPE excitation mode are as follows:

tbl_cba_gain[]	look-up-table for adaptive codebook gain
tbl_cbf_gain[]	look-up-table for fixed codebook gain
tbl_cbf_gain_dif []	look-up-table for fixed codebook gain difference
cba[]	adaptive codebook
prev_Gf	fixed codebook gain of the previous subframe

3.5.7.1.3 Decoding process

3.5.7.1.3.1 Shape decoder

This block describes the extraction of Adaptive codebook lag and the RPE parameters. The adaptive codebook lag is derived from the **shape_delay** in the following way:

```
lag = Lmax - Lmin - shape_delay
```

where Lmax and Lmin are the maximum and minimum lag, respectively, equal to 295 and 40. The decimation factor D and number of pulses Np are tabulated below. The RPE parameters, namely the RPE phase (rpe_phase) and the RPE amplitudes (rpe_amps) are derived as follows:

```
rpe_index = shape_index;
rpe_phase = rpe_index MOD D
rpe_index = rpe_index / D;
for (n = Np - 1; n >= 0; n--)
{
   rpe_amps[n]  = (rpe_index MOD 3) - 1;
   rpe_index   = rpe_index / 3;
}
```

3.5.7.1.3.2 Gain decoder

This block derives the scalar quantized adaptive and fixed codebook gains from the gain indices. The adaptive codebook gain Ga is determined by a table look-up on cba_gain[] (Table 3.88):

```
if (gain_indices[0] > 31)
{
   Ga = -1 * cba_gain[((64 - gain_indices[0]) - 1)];
}
else
{
   Ga = cba_gain[gain_indices[0]];
}
```

The decoding of the fixed codebook gain depends on the subframe under consideration. The gain vector is used for retrieving the fixed codebook gain Gf . For the first subframe in a frame the gain is decoded with:

```
Gf = cbf_gain[gain_indices[1]];
```

where cbf_gain[] is provided in Table 3.89. For all later subframes the gain is decoded using cbf_gain_dif[] provided in Table 3.90:

```
Gf = cbf_gain_dif[gain_indices[1]] * prev_Gf;
```

where prev_Gf is the decoded gain of the previous subframes.

3.5.7.1.3.3 Adaptive codebook generation

First the generation of the excitation signal due to the adaptive codebook will be described. The excitation for the adaptive codebook ya[n] is computed using the shape vector:

```
for (n = 0; n < sbfrm_size; n ++)
{
   ya[n] = cba[lag + n];
}
```

3.5.7.1.3.4 Fixed codebook generation

The excitation for the fixed codebook is computed in two steps. To generate the fixed codebook, we also need two extra parameters, namely D (the pulse spacing or the decimation factor) and Np, the number of pulses.

These values are dependent on the bitrate and are tabulated in Table 3.87.

Table 3.87 — Allocation of pulse spacing and number of pulses in RPE

RPE_Configuration	D	Np
0	8	5
1	8	5
2	5	6
3	4	6
4 ... 7	Reserved	

Once the phase and amplitudes are known, the amplitudes are placed on a regular grid. The amplitudes can have 3 different values: -1, 0 and 1. The excitation for the fixed codebook, yf, is computed using the RPE-formula:

```
for (n = 0; n < sbfrm_size; n++)
{
   yf[n] = 0;
}
for (n = 0; n < Np; n++)
{
   yf[phase + D*n] = amp[n];
}
```

3.5.7.1.3.5 Excitation generation

The excitation is the sum of adaptive and fixed excitation multiplied by their corresponding gains:

```
for (n = 0; n < sbfrm_size; n++)
{
   excitation[n] = Ga * ya[n] + Gf * yf[n];
}
```

Finally, the adaptive codebook has to be updated using the excitation. This is done by shifting the adaptive codebook by sbfrm_size entries and filling the empty entries with the calculated excitation.

```
for (n = sbfrm_size; n < Lmax; n++)
{
   cba[n-sbfrm_size] = cba[n];
}
for (n = 0; n < sbfrm_size; n++)
{
   cba[Lmax-1-n] = excitation[sbfrm_size - 1 - n];
}
```

The adaptive codebook is initialized by filling it with zeros.

Table 3.88 — Representation table for adaptive codebook gain

Scale factor: 2^17

Index	cba_gain	Index	cba_gain
0	12386	3	47147
1	27800	4	54827
2	38483	5	61669

Scale factor: 2^15

Index	cba_gain	Index	cba_gain
6	16993	15	30284
7	18520	16	31877
8	20021	17	33607
9	21493	18	35550
10	22938	19	37808
11	24396	20	40613
12	25834	21	44122
13	27292	22	48847
14	28767	23	55745

Scale factor: 2^8

Index	cba_gain	Index	cba_gain
24	528	28	5553
25	727	29	11107
26	1388	30	22214
27	2777	31	44426

Table 3.89 — Representation table for fixed codebook gain

Scale factor: 2^12

Index	cbf_gain	Index	cbf_gain
0	8192	4	26573
1	11578	5	34324
2	14877	6	44160
3	19909	7	55564

Scale factor: 2^9

Index	cbf_gain	Index	cbf_gain
8	8739	14	28624
9	10823	15	33682
10	13458	16	39118
11	16451	17	45372
12	19978	18	52626
13	24039	19	60243

Scale factor: 2^5

Index	cbf_gain	Index	cbf_gain
20	4327	26	10689
21	4954	27	13066
22	5700	28	16183
23	6551	29	21291
24	7649	30	32832
25	8977	31	reserved

Table 3.90 — Representation table for differential fixed codebook gain

Scale factor: 2^13

Index	cbf_gain_dif	Index	cbf_gain_dif
0	819	4	9544
1	3205	5	12763
2	5418	6	22118
3	7335	7	53248

3.5.7.2 Multi-pulse excitation generation tool

3.5.7.2.1 Tool description

Figure 3.7 shows the block diagram of the Multi-Pulse Excitation Generation tool. For clarity, the indexing for each subframe is dropped. It is noted that the excitation generation process is repeated every subframe. The excitation signal is constructed from a periodic component (adaptive codebook vector) and non-periodic component (fixed codebook vector) scaled by their respective gains. Both the adaptive and the fixed codebook vectors are decoded from **shape_delay, shape_positions** and **shape_signs**. The gains are decoded from three types of indices, **signal_mode**, **rms_index** and **gain_index**.

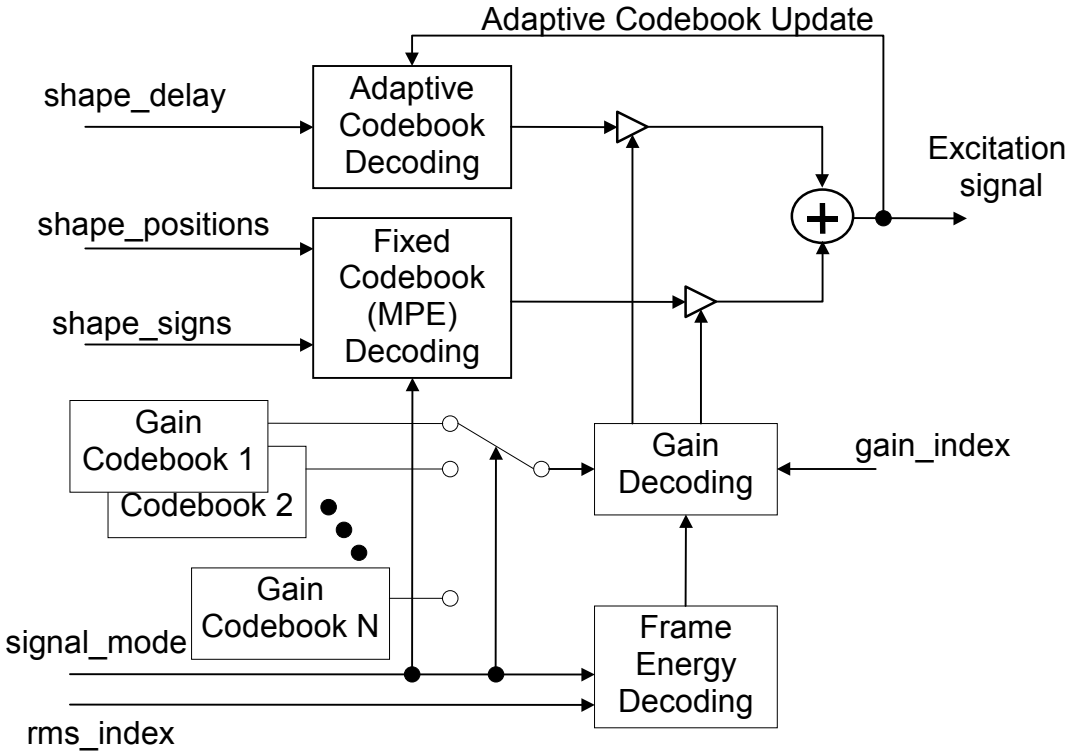


Figure 3.7 — Multi-pulse excitation generator

3.5.7.2.2 Definitions

Input

shape_delay[]: This array is of dimension *nrof_subframes* and contains the adaptive codebook lag.

shape_positions[]: This array is of dimension *nrof_subframes* and contains the pulse positions.

shape_signs[]: This array is of dimension *nrof_subframes* and contains the pulse signs.

gain_index[]: This array is of dimension *nrof_subframes* and contains the adaptive codebook gain index and the fixed codebook gain index.

rms_index: This field defines the index for the signal power.

signal_mode: This field contains the voiced/unvoiced flag.

int_Qlpc_coefficients[]: This is an array of dimension *lpc_order* and contains the quantized and interpolated LPC coefficients of one subframe.

Output

excitation[]: This array is of dimension sbfrm_size and contains the excitation signal. This signal is reconstructed from shape and gain vectors using the adaptive and fixed codebooks.

acb_delay: This field contains the decoded lag for the adaptive codevector.

adaptive_gain: This field contains the decoded gain for the adaptive codevector.

Configuration

lpc_order: This field indicates the order of LPC that is used.

sbfrm_size: This field indicates the number of samples in the subframe.

nrof_subframes: This field indicates the number of subframes.

The additional elements used in the MPE tool are as follows:

pacb[]	look-up-table of excitation for the periodic component
pos_tbl[][]	look-up-table of excitation for the non-periodic component
acb[]	decoded excitation signal as the periodic component
fcb[]	decoded excitation signal as the non-periodic component
ga	decoded gain for the periodic component
gf	decoded gain for the non-periodic component
qxnorm[]	decoded root mean squares of the speech signal
par[]	reflection coefficients converted from the *int_Qlpc_coefficients*[]
acb_energy	energy of *acb[]*
fcb_energy	energy of *fcb[]*
acb_delay	Integer part of the pitch lag
acb_frac	Fractional part of the pitch lag

3.5.7.2.3 Decoding process

3.5.7.2.3.1 Decoding of signal_mode

Signal_mode represents one of four modes for each frame. Modes 0 and 1 correspond to unvoiced and transition frames, respectively. Modes 2 and 3 correspond to voiced frames, and the latter indicates higher pitch periodicity than the former. This information is utilized in decoding the frame energy, the multi-pulse excitation and gains.

3.5.7.2.3.2 Decoding of frame energy

The root mean square (rms) value at last subframe is reconstructed using **signal_mode** and **rms_index**. The rms value is decoded in the μ-law scale. μ-law parameters are dependent on **signal_mode**. The rms values of other subframes are obtained by linear-interpolating the decoded rms values at the last subframe of the current and the previous frames. The quantized rms values are used for the gain decoding process.

```
delt = 1.0 / 64;
aa = 1.0 / log10(1.0 + mu_law);
bb = rms_max / mu_law;

pwk = aa * log10(1.0 + pqxnorm / bb);
qwk = delt*(rms_index+1);
for (i = 0; i < n_subframes; i++ )
{
   nwk = (qwk-pwk)*(i+1)/n_subframes + pwk;
```

```
   qxnorm[i] = bb * (pow((double)10.0,(nwk/aa)) - 1.0);
}
pqxnorm = qxnorm [n_subframes-1];
```

Table 3.91 — Value of rms_max and mu_law

Signal_mode	rms_max	mu_law
0	7932	1024
1, 2, 3	15864	512

3.5.7.2.3.3 Decoding of the adaptive codebook vector

The integer and the fractional parts of the pitch delay are obtained from **shape_delay**. The mapping tables between **shape_delay** and the two parts of the pitch delay are shown in Table 3.92 for the 8 kHz sampling rate and Table 3.93 for the 16 kHz sampling rate. The adaptive codebook vector *acb[n]* is computed by interpolating the past excitation signal *pacb[n]* at the decoded integer delay *acb_delay* and fraction *acb_frac*. The interpolation is done using an FIR filter *int_fil[k]* based on a Hamming-windowed sinc function. If the value of **shape_delay** is 255 for the 8 kHz sampling rate or 511 for the 16 kHz sampling rate, the output *acb[]* consists of all zero-valued samples. For the other indices, the output *acb[]* is produced by the following procedure:

```
for (n = 0; n < sbfrm_size;)
{
   tt += acb_frac;
   kt = acb_delay + tt / 6;
   tt = tt % 6;
   for (i = 0; i < kt && n < sbfrm_size; i++, n++)
   {
      for ( k = -iftap; k <= iftap; k++)
      {
         kk = (k+1) * 6 - tt;
         acb[n] += int_fil[abs(kk)] * pacb[pacb_size-(kt-i+k+1)];
         pacb[pacb_size+n] = acb[n];
      }
   }
}
```

where the filter coefficients for the 8 kHz and 16 kHz sampling rates are listed in Annex 3.C. The help parameters *iftap* and *pacb_size* are dependent on the sampling rate.

Table 3.92 — The mapping table between the shape_delay and the pitch delay for 8 kHz sampling rate

shape_delay	*acb_delay*	*acb_frac*
0 – 161	**shape_delay**/3+17	(2*__shape_delay__)%6
162 – 199	(__shape_delay__- 162)/2+71	(3*(__shape_delay__-162))%6
200 – 254	__shape_delay__-200+90	0
255	0	0

Table 3.93 — The mapping table between the shape_delay and the pitch delay for 16 kHz sampling rate

shape_delay	*acb_delay*	*acb_frac*
0 – 215	**shape_delay**/3+20	(2*__shape_delay__)%6
216 – 397	(__shape_delay__- 216)/2+92	(3*(__shape_delay__-216))%6
398 – 510	__shape_delay__-398+183	0
511	0	0

Table 3.94 — The number of interpolation filter taps and adaptive codebook size

Sampling Rate	*iftap*	*pacb_size*
8 kHz	6	150
16 kHz	11	306

3.5.7.2.3.4 Decoding of the fixed codebook vector

The fixed codebook vector contains several non-zero pulses and is represented by the pulse position and pulse amplitudes. The pulse positions *pul_pos[i]* are extracted from **shape_positions**. The pulse amplitudes *pul_amp[i]* are obtained from **shape_signs**.

```
for (i = num_pulse-1, k = 0; i >= 0; i--)
{
   for (j = 0; j < num_bit_pos[i]; j++)
   {
      pos_idx[i] |= ((shape_positions>>k) & 0x1) << j;
      k++;
   }
   pul_amp[i] = 1.0;
   if (((shape_signs >> (num_pulse-1-i)) & 0x1) == 1)
   {
      pul_amp[i] = -1.0;
   }
   pul_pos[i] = pos_tbl[i][pos_idx[i]];
}
```

Table 3.95 — The number of pulses for 8 kHz sampling rate

MPE_Configuration	num_pulse
0	3
1	4
2	5
3	5
4	6
5	7
6	6
7	7
8	8
9	9

MPE_Configuration	num_pulse
10	10
11	11
12	12
13	4
14	5
15	6
16	7
17	8
18	9
19	10

MPE_Configuration	num_pulse
20	11
21	12
22	8
23	9
24	10
25	11
26	12
27 ... 31	reserved

Table 3.96 — The number of pulses for 16 kHz sampling rate

MPE_Configuration	num_pulse
0, 16	5
1, 17	6
2, 18	7
3, 19	8
4, 20	9
5, 21	10
6, 22	11
7, 23	reserved

MPE_Configuration	num_pulse
8, 24	3
9, 25	4
10, 26	5
11, 27	6
12, 28	7
13, 29	8
14, 30	9
15, 31	10

where *num_pulse* is the number of pulses and is set from **MPE_Configuration** depending on the sampling rate. *num_bit_pos[i]* is the number of bits for encoding the *i*-th pulse position. *pos_tbl[i][j]* is the restriction table, which indicates the possible positions for each pulse. The table indicates the possible positions for each pulse. The table is set up in accordance with the combination of *subfrm_size*, *num_pulse* and *num_bit_pos[]* as follows:

```
step = subfrm_size / min_num_bit_pos;
for (i = 0; i < num_pulse; i++)
{
   m = 1 << (num_bit_pos[i]-min_num_bit_pos);
   for ( j = 0, k = 0; k < m)
   {
      ch[j] = i;
      k++;
      j += (long)((float)step/m + 0.5);
      j = j % step;
   }
}


for (i = 0; i < num_pulse; i++)
{
   for (l = 0, k = 0; k < step; k++)
   {
      if (i == ch[k])
      {
         for (j = 0; j < min_num_bit_pos; j++)
         {
            pos_tbl[i][l++] = k + step * j;
         }
      }
   }
}
```

For example, the pulse position table used in 6 kbit/s Mode II coder for the 8 kHz sampling rate is shown in Table 3.97.

Table 3.97 — The pulse position table for 6 kbit/s coder

Pulse number: I	Num_bit_pos[i]	Pulse positions: pos_tbl[i][j]
0	4	0,5,10,15,20,25,30,35, 40,45,50,55,60,65,70,75
1	4	1,6,11,16,21,26,31,36, 41,46,51,56,61,66,71,76
2	4	2,7,12,17,22,27,32,37 42,47,52,57,62,67,72,77
3	3	3,13,23,33,43,53,63,73
4	3	4,14,24,34,44,54,64,74
5	3	8,18,28,38,48,58,68,78
6	3	9,19,29,39,49,59,69,79

The fixed codebook vector *fcb[n]* is computed from *pul_pos[i]* and *pul_amp[i]* as follows:

```
for (n = 0; n < sbfrm_size; n++)
{
   fcb[n] = 0.0;
}
for (i = 0; i < num_pulse; i++)
{
   fcb[pul_pos[i]] = pul_amp[i];
}
```

If the integer delay *acb_delay* is less than the subframe size *sbfrm_size*, the fixed codebook vector signal *fcb[n]* is modified by zero-state-combfiltering as follows:

```
for (n = 0; n < sbfrm_size; n++)
{
   if (n - acb_delayt >= 0)
   {
      ix = fcb[n - acb_delay];
   }
   else
   {
      ix = 0.0;
   }
   fcb[n] += cga[signal_mode] * ix;
}
```

where *cga[4] = {0.0, 0.0, 0.6, 0.8}* are the gains of the comb-filter.

3.5.7.2.3.5 Decoding of the adaptive and the fixed codebook gains

The **gain_index** is converted to the normalized gains *nga, ngf* for the adaptive and the fixed codebook vectors by looking up the gain table. The gain table is changed in accordance with **signal_mode**, *sbfrm_size* and sampling rate. The gain tables for this tool are given in Annex 3.C. The adaptive codebook gain *ga* and the fixed codebook gain *gf* are calculated as follows:

```
ga = nga * sqrt (norm / acb_energy);
gf = ngf * sqrt (norm / fcb_energy);
```

where *acb_energy* and *fcb_energy* are the energies for the adaptive codebook and fixed codebook vectors, respectively. The *norm* is the normalization factor.

```
norm = (qxnorm*subfrm_size)*(qxnorm*subfrm_size);
for(i = 0; i < lpc_order; i++)
{
   norm *= (1 - par[i] * par[i]);
}
```

where *par[]* are the reflection coefficients and are calculated from the LP coefficients *int_Qlpc_coefficients[]*. *qxnorm* is the quantized subframe energy, which is decoded from the **rms_indx** (see subclause 3.5.7.2.3.2).

3.5.7.2.3.6 Excitation signal generation

The excitation signal (*excitation[]*) is calculated by summing *acb[]* and *fcb[]* scaled by *ga and gf* respectively.

```
for (i = 0; i < sbfrm_size; i++)
```

```
{
   excitation[i] = ga*acb[i] + gf*fcb[i];
}
```

3.5.7.2.3.7 Updating the adaptive codebook

The adaptive codebook is updated for the decoding process at the next frame by the generated excitation signal *excitation[]* as follows:

```
for (i = 0; i < pacb_size - sbfrm_size; i++)
{
   pacb[i] = pacb[sbfrm_size+i];
}


for (i = 0; i < sbfrm_size; i++)
{
   pacb[pacb_size-sbfrm_size+i] = excitation[i];
}
```

3.5.7.3 Bitrate scalable multi-pulse excitation generation tool

3.5.7.3.1 Tool description

The bitrate scalable decoder is realized using the bitrate scalable Multi-Pulse Excitation tool, which consists of the Multi-Pulse Excitation tool and the enhancement excitation decoding tool. This scalability enhancement is allowed only for the Mode II coder. The enhancement excitation signal is reconstructed by retrieving **shape_enh_positions, shape_enh_ signs, gain_enh_index** and utilizing the decoded multi-pulse excitation signal in the Multi-Pulse Excitation tool.

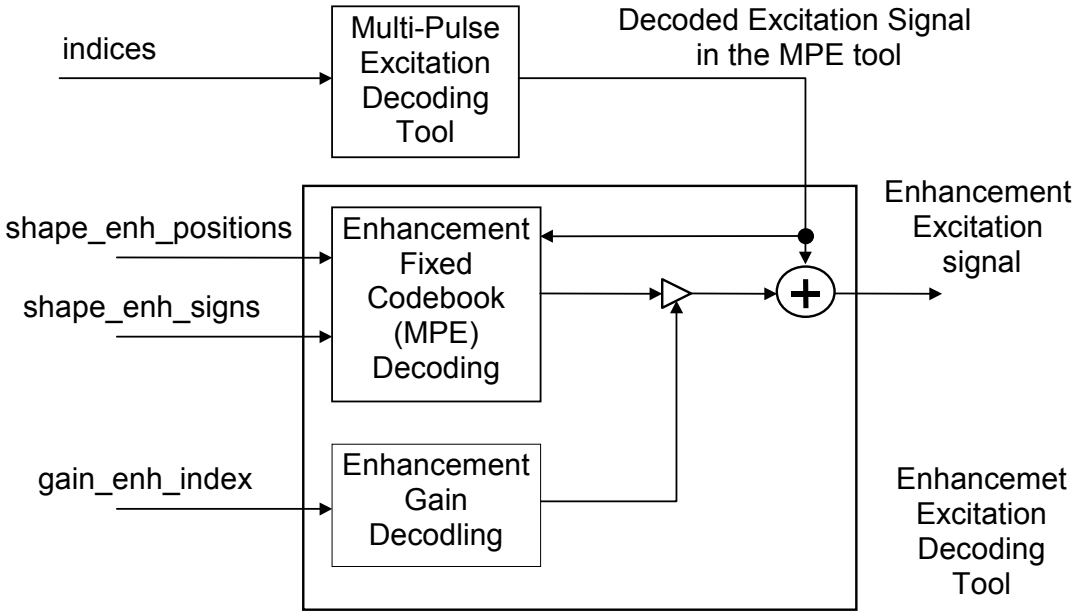


Figure 3.8 — Bitrate scalable multi-pulse excitation generation process

3.5.7.3.2 Definitions

Input

shape_delay[]: This array is of dimension *nrof_subframes* and contains the adaptive codebook lag.

shape_positions[]: This array is of dimension *nrof_subframes* and contains the pulse positions.

shape_signs[]: This array is of dimension *nrof_subframes* and contains the pulse signs.

shape_enh_positions[]: This array is of dimension *nrof_subframes* and contains the pulse positions.

shape_enh_signs[]: This array is of dimension *nrof_subframes* and contains the pulse signs.

gain_enh_index[]: This array is of dimension *nrof_subframes* and contains the adaptive codebook gain index and the fixed codebook gain index.

int_Qlpc_coefficients[]: This is an array of dimension *lpc_order* and contains the quantized and interpolated LPC coefficients of one subframe.

Output

enh_excitation[]: This array is of dimension sbfrm_size and contains the enhancement excitation signal. This signal is reconstructed from shape and gain vectors using the fixed codebook.

Configuration

lpc_order: This field indicates the order of LPC that is used.

sbfrm_size: This field indicates the number of samples in the subframe.

nrof_subframes: This field indicates the number of subframes.

The additional elements used in the Bitrate Scalable MPE tool are as follows:

pos_tbl[][]	look-up-table of excitation for the non-periodic component
enh_fcb[]	decoded excitation signal as the non-periodic component
ge	decoded gain for the non-periodic component
enh_fcb_energy	energy of *enh_fcb[]*

3.5.7.3.3 Decoding process

3.5.7.3.3.1 Decoding of the enhancement fixed codebook vector

The enhancement fixed codebook vector also consists of several non-zero pulses. The pulse positions and amplitudes are extracted from **shape_enh_positions, shape_enh_signs** by the same decoding algorithm as the fixed codebook. The enhancement fixed codebook vectors *enh_fcb[n]* is computed from *pul_pos[i]* and *pul_amp[i]* as follows:

```
for (i = num_pulse_enh-1, k = 0; i >= 0; i--)
{
   for (j = 0; j < num_bit_pos[i]; j++)
   {
      pos_idx[i] |= ((shape_enh_positions>>k)&0x1) << j;
      k++;
   }
   pul_amp[i] = 1.0;
   if (((shape_enh_signs >> (num_pulse_enh-1-i)) & 0x1) == 1)
   {
      pul_amp[i] = -1.0;
   }
   pul_pos[i] = pos_tbl[i][pos_idx[i]];
}

for (n = 0; n < sbfrm_size; n++)
{
   enh_fcb[n] = 0.0;
}
for (i = 0; i < num_pulse_enh; i++)
{
   enh_fcb[pul_pos[i]] = pul_amp[i];
}
```

Table 3.98 — Definition of num_pulse_enh

sbfrm_size	*num_pulse_enh*
40	2
80	4

The pulse position table used in the enhancement tool is adaptively controlled so that the enhancement pulses stand at different positions from those of the pulses selected in the core excitation. The pulse position table is temporarily generated by the same decoding algorithm as the fixed codebook. The temporary pulse position table is modified as follows,

```
for (n = 0; n < num_enh; n++)
{
   for (i = num[n]-1, k = 0; i >= 0; i--)
   {
      pul_loc = 0;
      for (j = 0; j < bit_pos[i]; j++)
      {
         pul_loc |= ((idx[n]>>k)&0x1)<<j;
         k++;
      }
      pul_loc = chn_pos[i*len+pul_loc];
      for (l = 0; l < 10; l++)
      {
         for (m = 0; m < (1 << bit_pos_org[l]); m++)
         {
            if (pul_loc == chn_pos_org[l*len+m])
            {
               chn_ctr[l]++;
               break;
            }
         }
      }
   }

   for (i = 0; i < 10; i++)
   {
      ctr_tmp[i] = chn_ctr[i];
   }
   for (i = 0; i < num[n+1]; i++)
   {
      min_ctr = len;
      for (j = 0; j < 10; j++)
      {
         if (ctr_tmp[j] < min_ctr)
         {
            min_ctr = ctr_tmp[j];
            min_chn = j;
         }
      }
      ctr_tmp[min_chn] = len;
      bit_pos[i] = bit_pos_org[min_chn];
      for (j = 0; j < (1<<bit_pos_org[min_chn]); j++)
      {
         chn_pos[i*len+j] = chn_pos_org[min_chn*len+j];
      }
   }
}

for (i = 0; i < num[num_enh]; i++)
{
   bit[i] = bit_pos[i];
   for (j = 0; j < (1<<bit[i]); j++)
   {
      pos_tbl[i*len+j] = chn_pos[i*len+j];
   }
}
```

The temporary pulse position tables used in the enhancement excitation tool is shown in Table 3.99 and Table 3.100. The table changes depending on the subframe length.

Table 3.99 — The temporary pulse position table for 80-sample subframe

Pulse number: i	bit_pos_org[i]	Pulse positions: chn_pos_org[i][j]
0	3	0,10,20,30,40,50,60,70
1	3	1,11,21,31,41,51,61,71
2	3	2,12,22,32,42,52,62,72
3	3	3,13,23,33,43,53,63,73
4	3	4,14,24,34,44,54,64,74
5	3	5,15,25,35,45,55,65,75
6	3	6,16,26,36,46,56,66,76
7	3	7,17,27,37,47,57,67,77
8	3	8,18,28,38,48,58,68,78
9	3	9,19,29,39,49,59,69,79

Table 3.100 — The temporary pulse position table for 40-sample subframe

Pulse number: i	bit_pos_org[i]	Pulse positions: chn_pos_org[i][j]
0	2	0,10,20,30
1	2	1,11,21,31
2	2	2,12,22,32
3	2	3,13,23,33
4	2	4,14,24,34
5	2	5,15,25,35
6	2	6,16,26,36
7	2	7,17,27,37
8	2	8,18,28,38
9	2	9,19,29,39

3.5.7.3.3.2 Decoding of the enhancement fixed codebook gain

The **gain_enh_index** is converted to the normalized gain *nge* for the enhancement fixed codebook vector by looking up the gain table. The gain table is changed in accordance with **signal_mode** and is given in Annex 3.C. The enhancement fixed codebook gain *ge* are calculated as follows:

```
ge = nge * sqrt (norm / enh_fcb_energy);
```

where the *enh_fcb_energy* is the energy for the enhancement fixed codebook vector. The *norm* is the normalization factor.

3.5.7.3.3.3 Enhanced excitation signal generation

The enhanced excitation signal (enh_*excitation[]*) is calculated by adding the enhancement codebook vector to excitation signals.

```
for (i = 0; i < sbfrm_size; i++)
{
   enh_excitation[i] = excitation[i] + ge*enh_fcb[i];
}
```

3.5.7.4 Bandwidth scalable multi-pulse excitation generation tool

3.5.7.4.1 Tool description

The excitation decoding process in the bandwidth scalable decoder is achieved by the bandwidth scalable Multi-Pulse Excitation tool, which consists of the Multi-Pulse Excitation tool and the bandwidth extension tool as shown in Figure 3.9. This scalability enhancement is allowed only for the Mode II coder. The scalable excitation signal is reconstructed by retrieving **shape_bws_positions, shape_bws_ signs** and **gain_bws_index** and utilizing the decoded outputs for the 8 kHz sampling rate, namely the integer and the fractional parts of the pitch delay and the fixed codebook vector. These outputs are generated at the MPE decoding tool (subclause 3.5.7.2) or the bitrate scalable MPE decoding tool (subclause 3.5.7.3).

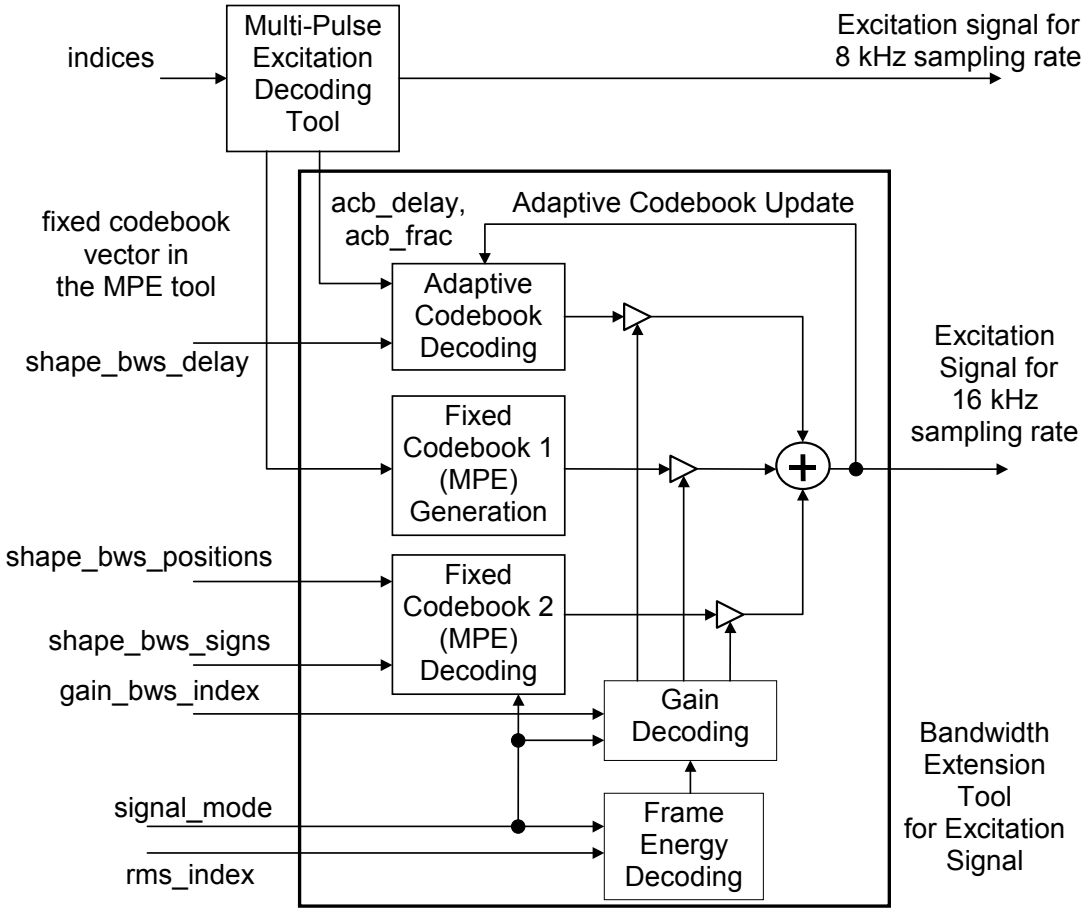


Figure 3.9 — Bandwidth scalable multi-pulse excitation generation process

3.5.7.4.2 Definitions

Input

shape_bws_delay[]: This array is of dimension *nrof_subframes_bws* and contains the adaptive codebook lag.

shape_bws_positions[]: This array is of dimension *nrof_subframes_bws* and contains the pulse positions.

shape_bws_signs[]: This array is of dimension *nrof_subframes_bws* and contains the pulse signs.

gain_bws_index[]: This array is of dimension *nrof_subframes_bws* and contains the adaptive codebook gain index and the fixed codebook gain index.

rms_index: This field defines the index for the signal power.

signal_mode: This field contains the voiced/unvoiced flag.

int_Qlpc_coefficients[]: This is an array of dimension *lpc_order* and contain the quantized and interpolated LPC coefficients of one subframe.

Output

excitation[]: This array is of dimension sbfrm_size and contains the excitation signal. This signal is reconstructed from shape and gain vectors using the adaptive and fixed codebooks.

acb_delay: This field contains the decoded lag for the adaptive codevector.

adaptive_gain: This field contains the decoded gain for the adaptive codevector.

Configuration

lpc_order: This field indicates the order of LPC that is used.

sbfrm_size: This field indicates the number of samples in the subframe in the bandwidth extension tool.

nrof_subframes_bws: This field indicates the number of subframes in the bandwidth extension tool.

The additional elements used in the Bandwidth Scalable MPE tool are as follows:

pacb[]	look-up-table of excitation for the periodic component
pos_tbl[][]	look-up-table of excitation for the non-periodic component
acb[]	decoded excitation signal as the periodic component
fcb1[]	decoded excitation signal as the non-periodic component
fcb2[]	decoded excitation signal as the non-periodic component
ga	decoded gain for the periodic component
gn	decoded gain for the non-periodic component
gf	decoded gain for the non-periodic component
qxnorm[]	decoded root mean squares of the speech signal
par[]	reflection coefficients converted from the *int_Qlpc_coefficients*[]
acb_energy	energy of *acb[]*
fcb1_energy	energy of *fcb1[]*
fcb2_energy	energy of *fcb2[]*
acb_delay_wb	Integer part of the pitch lag
acb_frac_wb	Fractional part of the pitch lag

3.5.7.4.3 Decoding process

For the Mode II decoder with bandwidth scalability, the excitation signal at 16 kHz sampling is constructed from a periodic component (adaptive codebook vector) and two non-periodic components (fixed codebook vector 1 and 2) scaled by their respective gains.

3.5.7.4.3.1 Decoding of signal_mode

Signal_mode is also utilized in decoding the frame energy, the multi-pulse excitation and gains in this decoding process.

3.5.7.4.3.2 Decoding of frame energy

The decoding procedure is same as the MPE decoding tool.

3.5.7.4.3.3 Decoding of the adaptive codebook vector

The integer and the fractional parts of the pitch delay are obtained from **shape_delay** and **shape_bws_delay**.

The *acb_delay* and *acb_frac* in the 8 kHz sampling rate are decoded at the MPE tool and are fed to the bandwidth extension tool for the excitation signal. The 8 kHz sampling rate parameters are converted to the 16 kHz sampling rate parameters *acb_delay_wb*, *acb_frac_wb* in 16 kHz sampling rate as follows:

```
op_delay_wb = 2 * acb_delay
if (acb_frac != 0)
{
   op_delay_wb++;
}

if (op_delay_ wb == 0)
{
   op_idx_wb = 778;
}
else
{
   op_idx_wb = (op_delay_wb - 32) * 3 + 2;
}

st_idx_wb = op_idx_wb - 4;
if (st_idx_wb < 0)
{
   st_idx_wb = 0;
}
if ((st_idx_wb + 7) >= 778)
{
   st_idx_wb = 778 - 8;
}

if (op_idx_wb == 778)
{
   acb_idx_wb = 778;
}
else
{
   acb_idx_wb = st_idx_wb + shape_bws_delay;
}
```

The mapping table between acb_idx_wb and the pitch delay parameters acb_delay_wb, acb_frac_wb is shown in Table 3.101. The adaptive codebook vector *acb[n]* is computed by interpolating the past excitation signal *pacb[n]* at the decoded integer delay acb_delay_wb and fraction acb_frac_wb. The interpolation is done using an FIR filter *int_fil[k], k=0,...,66* based on a Hamming windowed sinc function. If the value of **shape_delay** is 255 or the value of **shape_bws_delay** is 768, the output *acb[]* consists of all zero-valued samples. For other combinations of the indices, the output *acb[]* is produced by the following procedure:

```
for (n = 0; n < sbfrm_size;)
{
   tt += acb_frac_wb;
   kt = acb_delay_wb + tt / 6;
   tt = tt % 6;
   for (i = 0; i < kt && n < sbfrm_size; i++, n++)
   {
      for (k = -11; k <= 11; k++ )
      {
         kk = (k+1) * 6 - tt;
         acb[n] += int_fil [abs(kk)] * pacb [306-(kt-i+k)];
         pacb[306+n] = acb[n];
      }
   }
}
```

where the filter coefficients for the 16 kHz sampling rate are listed in Annex 3.C.

Table 3.101 — The mapping table between the shape_delay and the pitch delay

acb_idx_wb	acb_delay_wb	acb_frac_wb
0, 1	32	(2*(acb_idx_wb+1)%6
2 - 777	32+2*(acb_idx_wb-2)/6	(2*(acb_idx_wb-2))%6
778	0	0

3.5.7.4.3.4 Decoding of the fixed codebook vector 1

The fixed codebook vector 1 fcb1[n] is obtained by sampling-rate conversion of the fixed codebook vector nb_fcb[n] used in the MPE tool or the bitrate scalable MPE tool as follows:

```
for (n = 0; n < sbfrm_size/2; n++)
{
   fcb1[2*n] = nb_fcb[n];
   fcb1[2*n+1] = 0;
}
```

3.5.7.4.3.5 Decoding of the fixed codebook vector 2

The fixed codebook vector 2 contains several non-zero pulses and is represented by the pulse position and pulse amplitudes. The pulse positions *pul_pos[i]* are extracted from **shape_bws_positions**. The pulse amplitudes *pul_amp[i]* are obtained from **shape_bws_signs**.

```
for (i = num_pulse_bws - 1, k = 0; i >= 0; i--)
{
   for (j = 0; j < num_bit_pos[i]; j++)
   {
      pos_idx[i] |= ((shape_bws_positions >> k) & 0x1) << j;
      k++;
   }
   pul_amp[i] = 1.0;

   if (((shape_bws_signs >> (num_pulse_bws-1-i)) & 0x1) == 1)
   {
      pul_amp[i] = -1.0;
   }
   pul_pos[i] = pos_tbl[i][pos_idx[i]];
}
```

where *num_pulse_bws* is the number of pulses and is one of 6, 8, 10, 12. The choice is dependent on BWS_configuration. *num_bit_pos[i]* is the number of bits for encoding the *i*-th pulse position. *pos_tbl[i][j]* is the restriction table, which indicates the possible positions for each pulse. The table indicates the possible positions for each pulse.

Table 3.102 — Definition of num_pulse_bws

BWS_configuration	*num_pulse_bws*
0	6
1	8
2	10
3	12

The pulse position tables for each number of pulses are also set up in accordance with the combination of *subfrm_size*, *num_pulse* and *num_bit_pos[]* by the same procedure as the MPE tool.

The fixed codebook vector *fcb2[n]* is computed from *pul_pos[i]* and *pul_amp[i]* as follows:

```
for (n = 0; n < sbfrm_size; n++)
{
   fcb2[n] = 0.0;
}
for (i = 0; i < num_pulse_bws; i++)
{
   fcb2[pul_pos[i]] = pul_amp[i];
}
```

If the integer delay *acb_delay_wb* is less than the subframe size *sbrm_size*, the fixed codebook vector signal *fcb2[n]* is modified by zero-state-comb filtering as follows:

```
for (n = 0; n < sbfrm_size; n++)
{
   if (n - acb_delayt >= 0)
   {
      ix = fcb2[n - acb_delay];
   }
   else
   {
      ix = 0.0;
   }
```

```
   fcb2[n] += cga[signal_mode] * ix;
}
```

where *cga[4] = {0.0, 0.0, 0.6, 0.8}* are the gains of the comb-filter.

3.5.7.4.3.6 Decoding of the adaptive and the fixed codebook gains

The **gain_bws_index** is converted to two indices.

```
qga_idx = gain_bws_index >> 7;
qgc_idx = gain_bws_index - (gain_bws_index << 7);
```

The *qga_idx* is converted to the normalized gains *nga, ngf* for the adaptive and the fixed codebook vector 2 by looking up the gain table. The *qgc_idx* is converted to the normalized gain *ngn* for the fixed codebook vector 1 by looking up the gain table. The gain table is changed in accordance with the **signal_mode** and is given in Annex 3.C. The adaptive codebook gain *ga* , the fixed codebook 1 gain gn and the fixed codebook 2 gain *gf* are calculated as follows:

```
ga = nga * sqrt( norm / acb_energy );
gn = ngn * sqrt( norm / fcb1_energy );
gf = ngf * sqrt( norm / fcb2_energy );
```

where the *acb_energy, fcb2_energy* and *fcb1_energy* are the energies for the adaptive codebook and the two fixed codebook vectors. The *norm* is the normalization factor.

```
norm = (qxnorm*subfrm_size) * (qxnorm*subfrm_size);
for (i = 0; i < lpc_order; i++)
{
   norm *= (1 - par[i] * par[i]);
}
```

where *par[]* are the reflection coefficients and are calculated from the LP coefficients *int_Qlpc_coefficients*[]. The *qxnorm* is the quantized subframe energy and is decoded from the **rms_indx**.

3.5.7.4.3.7 Excitation signal generation

The excitation signal (*excitation[]*) is calculated by summing *acb[], fcb1[]* and *fcb2[n]* scaled by *ga, gn and gf* respectively.

```
for (i = 0; i < sbfrm_size; i++)
{
   excitation[i] = ga*acb[i] + gn*fcb1[i] + gf*fcb2[i];
}
```

3.5.7.4.3.8 Updating the adaptive codebook

The adaptive codebook is updated for the decoding process at the next frame by the generated excitation signal *excitation[]* as follows:

```
for (i = 0; i < 306-sbfrm_size; i++)
{
   pacb[i] = pacb[sbfrm_size+i];
}


for (i = 0; i < sbfrm_size; i++)
{
   pacb[306-sbfrm_size+i] = excitation[i];
}
```

3.5.8 CELP LPC synthesis filter

3.5.8.1 Tool description

The CELP LPC synthesis filter constructs the synthesized signal from the LPC coefficients and the excitation signal for each subframe.

3.5.8.2 Definitions

Input

excitation[]: This array contains the excitation signal for one subframe.

int_Qlpc_coefficients[]: This array of dimension *lpc_order* contains the quantized and interpolated LPC coefficients.

Output

synth_signal[]: The excitation signal, excitation[], is fed through the synthesis filter using the LPC coefficients of *int_Qlpc_coefficients*[]. The dimension of this array is *lpc_order*.

Configuration

lpc_order: This field contains the order of LPC used.

sbfrm_size: This field contains the number of samples in the subframe.

3.5.8.3 Decoding process

Using the interpolated LPC coefficients of one subframe, the excitation signal is fed through the following filter:

$$H_S(z) = \frac{1}{\widehat{A}(z)} = \frac{1}{1 - \sum_{k=1}^{lpc\_order} \widehat{a}_k \cdot z^{-k}}$$

where $\widehat{A}(z)$ is the LPC inverse filter using the quantized LPC coefficients. The coefficient $\widehat{a}_k$ is the *k*-th LPC coefficient (*int_Qlpc_coefficients*[k-1]). The output of the inverse filter is the reconstructed speech. The LPC order is fixed to 10 and 20 for the 8 kHz and 16 kHz sampling rate, respectively.

The following algorithm is an implementation of the above filter.

```
for (n = 0; n < sbfrm_size; n++)
{
   tmp = excitation[n];
   for (k = 0; k < lpc_order; k++)
   {
      tmp = tmp + Filter_states[k] * int_Qlpc_coefficients[k];
   }
   synth_signal[n] = tmp;
   for (k = lpc_order-1; k > 0; k--)
   {
      Filter_states[k] = Filter_states[k-1];
   }
   Filter_states[0] = synth_signal[n];
}
```

The array Filter_states is initially set to zero.

3.5.9 CELP silence compression tool

3.5.9.1 Tool description

The silence compression tool comprises a Voice Activity Detection (VAD) module, a Discontinuous Transmission (DTX) unit and a Comfort Noise Generator (CNG) module. The tool encodes/decodes the input signal at a lower bitrates during the non-active (silent) frames. During the active-voice (speech) frames, MPEG-4 CELP encoding and decoding are used. A block diagram of the CELP codec system with the silence compression tool is depicted in Figure 3.10.

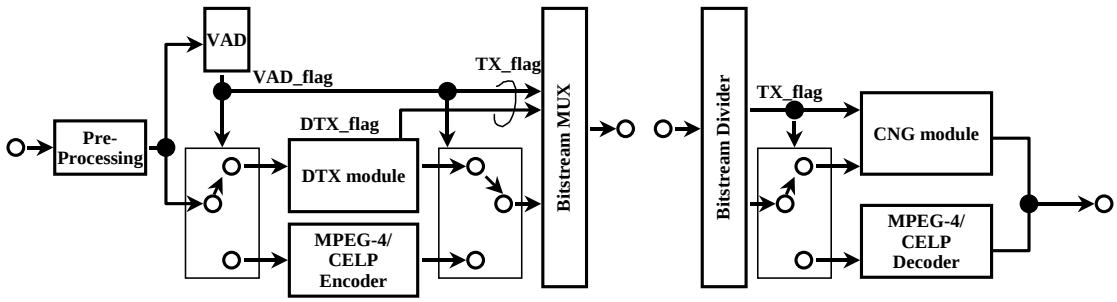


Figure 3.10 — Block diagram of the codec system with the silence compression tool

At the transmission side, the DTX module encodes the input speech during the non-active frames. During the active-voice frames, the MPEG-4 CELP encoder is used. The voice activity flag (VAD_flag) indicating a non-active frame (VAD_flag=0) or an active-voice frame (VAD_flag=1) is determined from the input speech by the VAD module. During the non-active frames, the DTX module detects frames where the input characteristics change (DTX_flag=1 and 2: Change, DTX_flag=0: No Change). When a change is detected, the DTX module encodes the input speech to generate a SID (Silence Insertion Descriptor) information. The VAD_flag and the DTX_flag are sent together as a TX_flag to the decoder to keep synchronization between the encoder and the decoder.

At the receive side, the CNG module generates comfort noise based on the SID information during the non-active frames. During the active-voice frames, the MPEG-4 CELP decoder is used instead. Either the CNG module or the MPEG-4 CELP decoder is selected according to the TX_flag.

The SID information and the TX_flag are transmitted only when a change of the input characteristics is detected. Otherwise only the TX_flag is transmitted during non-active frames.

3.5.9.2 Definitions

CNG: Comfort Noise Generation

Coding mode: "I" for the RPE and "II" for the MPE (see Table 3.1)

DTX: Discontinuous Transmission

LP: Linear Prediction

LPCs: LP Coefficients

MPE: Multi-Pulse Excitation

MPE_Configuration: see subclause 3.4

RMS: root mean square

RPE: Regular-Pulse Excitation

RPE_Configuration: see subclause 3.4

SID: Silence Insertion Descriptor

SID frame: frame where the SID information is sent/received

signal mode: mode determined based on the average pitch prediction gain (see subclause 3.4)

VAD: Voice Activity Detection

3.5.9.3 Decoding process

3.5.9.3.1 Transmission payload

There are four types of transmission payloads depending on the VAD/DTX decision. A TX_flag indicates the type of transmission payload and is determined by the VAD_flag and the DTX_flag as shown in Table 3.103. When the TX_flag indicates a active-voice frame (TX_flag = 1), information generated by the MPEG-4 CELP encoder and the TX_flag are transmitted. When the TX_flag indicates a transition frame between an active-voice frame and a non-active frame, or a non-active frame in which the spectral characteristics of the input signal changes (TX_flag = 2), High-Rate (HR) SID information and the TX_flag are transmitted to update the CNG parameters. When the TX_flag indicates a non-active frame in which the frame power of the input signal changes (TX_flag = 3), Low-Rate (LR) SID information and the TX_flag are transmitted. Other non-active frames are categorized into the fourth type of the TX_flag (TX_flag = 0). In this case, only the TX_flag is transmitted. Examples of the TX_flag change according to the VAD_flag and the DTX_flag are shown in Table 3.104.

Table 3.103 — Relation among flags for the silence compression tool

Flags	Active-voice	Non-active		
VAD_flag	1	0		
DTX_flag	-	0	1	2
TX_flag	1	0	2	3

Table 3.104 — Examples of the TX_flag change according to the VAD_flag and the DTX_flag

Frame #	...k-5	k-4	k-3	k-2	k-1	k	k+1	k+2	k+3	k+4	k+5	k+6	k+7
VAD_flag	...1	1	1	1	0	0	0	0	0	0	0	0	0...
DTX_flag	...-	-	-	-	1	0	0	0	1	0	0	0	2...
TX_flag	...1	1	1	1	2	0	0	0	2	0	0	0	3...

active-voice period	non-active period

3.5.9.3.2 LSP transmission

In case the silence compression tool is used in combination with FineRate Control enabled, a CELP frame with LPC_present = 1 and interpolation_flag = 0 must be transmitted in the first voice-active frame following a non-active frame. Voice-active frames are signaled by TX_flag = 1, non-active frames are signaled by TX_flag = 0, 2 or 3.

3.5.9.3.3 CNG module

Figure 3.11 depicts a structure of the CNG module that generates the comfort noise. The comfort noise is generated by filtering an excitation with an LP synthesis filter in a similar manner as with voice-active speech signals. The post filter may be used to improve the coding quality. The excitation is given by adding a multi-pulse excitation or a regular pulse excitation and a random excitation, scaled by their corresponding gains. The excitations are generated based on a random sequence independent of the SID information. The coefficients for the LP synthesis filter and gains are calculated from LSPs and a RMS value (frame energy) respectively, which are received as the SID information. The LSPs and the RMS are smoothed in order to improve the coding quality for the noisy input speech. The CNG module uses the same frame and subframe sizes as those in active speech frames. Processing in each part is described in the following subclauses.

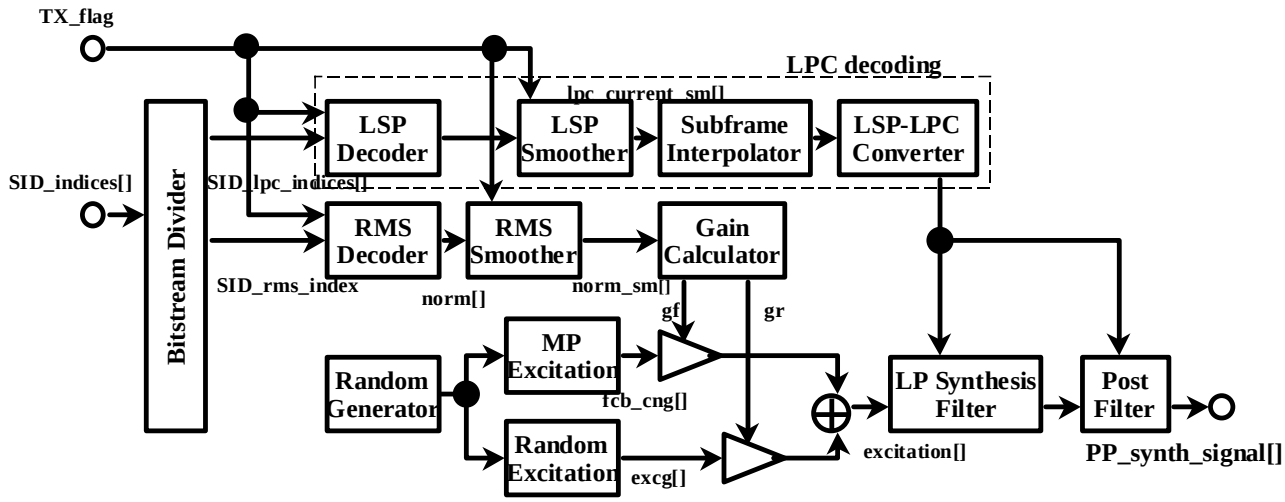


Figure 3.11 — CNG module

3.5.9.3.3.1 Definitions

Input

TX_flag	This field contains the transmission mode.
SID_lpc_indices[]	This array contains the packed LP indices. The dimension is 3, 5 or 6 (see Table 3.72).
SID_rms_index	This field contains the RMS index.

Output

PP_synth_signal[] This array contains the post-filtered (enhanced) speech signal. The dimension is *sbfrm_size*. The following are help elements used in the CNG module:

lpc_order: the order of LP

sbfrm_size: the number of samples in a subframe

n_subframe: the number of subframes in a frame

int_Q_lpc_coefficients[]*:* interpolated LPCs (see subclause 3.5.8.2).

3.5.9.3.3.2 LSP decoder

LSP *lpc_current*[] is decoded from the LSP indices *SID_lpc_indices*[]. The decoding process is identical to that described in subclause 3.5.6 with the following exceptions:

(1) A sub-set of the *lpc_indices*[] is transmitted to the decoder. A relation between the transmitted LSP indices *SID_lpc_indices*[] and the LSP indices for MPEG-4 CELP, *lpc_indices*[] is shown in Table 3.105.

(2) The decoding process is not carried out for the untransmitted indices.

Table 3.105 — LSP index relation between the silence compression tool and MPEG-4 CELP

Coding mode	Sampling rate [kHz]	Band width scalability	Silence compression tool	MPEG-4 CELP
I (RPE)	16	Off	SID_lpc_indices[0]	lpc_indices[0]
			SID_lpc_indices[1]	lpc_indices[1]
			SID_lpc_indices[2]	lpc_indices[2]
			SID_lpc_indices[3]	lpc_indices[3]
			SID_lpc_indices[4]	lpc_indices[5]
			SID_lpc_indices[5]	lpc_indices[6]
II (MPE)	8	On, Off	SID_lpc_indices[0]	lpc_indices[0]
			SID_lpc_indices[1]	lpc_indices[1]
			SID_lpc_indices[2]	lpc_indices[2]
	16	On	SID_lpc_indices[3]	lpc_indices[5]
			SID_lpc_indices[4]	lpc_indices[6]
			SID_lpc_indices[5]	lpc_indices[7]
			SID_lpc_indices[6]	lpc_indices[8]
		Off	SID_lpc_indices[0]	lpc_indices[0]
			SID_lpc_indices[1]	lpc_indices[1]
			SID_lpc_indices[2]	lpc_indices[2]
			SID_lpc_indices[3]	lpc_indices[3]
			SID_lpc_indices[4]	lpc_indices[5]
			SID_lpc_indices[5]	lpc_indices[6]

3.5.9.3.3.3 LSP smoother

Smoothed LSPs *lsp_current_sm*[] are updated using the decoded LSPs *lsp_current*[] in each frame as:

$$lsp\_current\_sm[i] = \begin{cases} 0.875\ lsp\_current\_sm[i] + 0.125\ lsp\_current[i], & TX\_flag = 2 \\ 0.875\ lsp\_current\_sm[i] + 0.125\ lsp\_current\_sid[i], & TX\_flag = 0\ or\ 3 \end{cases}$$

where *i=0,...,lpc_order-1* and *lsp_current_sid*[] is *lsp_current*[] in the last SID frame. At the beginning of every non-active period, *lsp_current_sm*[] is initialized with *lsp_current*[] at the end of the previous active-voice period.

3.5.9.3.3.4 LSP interpolation and LSP-LPC conversion

LPCs for the LP synthesis, *int_Q_lpc_coefficients*[] are calculated from the smoothed LSPs *lpc_current_sm*[] using the LSP interpolation with the stabilization and the LSP-to-LPC conversion. These processes are described in subclause 3.5.6. A common buffer for the previous frame *lsp_previous*[] is used in both the silence compression tool and in CELP coding for active speech frames.

3.5.9.3.3.5 RMS decoder

The RMS of the input speech *qxnorm* in each subframe is reconstructed using *SID_rms_index* in the same process as described in subclause 3.5.7.2.3.2 with the exception that the μ -law parameters are independent of the signal mode and set as *rms_max=7932* and *mu_law=1024*.

The reconstructed RMS of the input speech is converted into the RMS of the excitation signal (*norm*) using the reflection coefficients *par*[] as follows and used for the gain calculation:

```
norm = (qxnorm*subfrm_size)*(qxnorm*subfrm_size);
for (i = 0; i < lpc_order; i++)
{
    norm *= (1 - par[i] * par[i])* αs;
}
```

where *par*[] is calculated from the LPCs *int_Qlpc_coefficients*[] and a scaling factor α_s is *0.8*.

3.5.9.3.3.6 RMS smoother

A smoothed RMS *norm_sm* is updated using *norm* in each subframe as follows:

$$norm\_sm = \begin{cases} 0.875\ norm\_sm + 0.125\ norm[subnum] & for\ TX\_flag = 2\ or\ 3 \\ 0.875\ norm\_sm + 0.125\ norm\_sid & for\ TX\_flag = 0 \end{cases}$$

where *subnum* is the current subframe number ranging from 0 to *n_subframe-1* and *norm_sid* is *norm*[*n_subframe-1*] in the last SID frame. In the first frame of every non-active period, *norm_sm* is set to *norm.* During the first 40 msec of the non-active period, *norm_sm* is initialized with *norm* [*subnum*], *w*hen TX_flag = 2 or 3 and

$$\left| 20\log_{10} norm\_sid - 20\log_{10} norm[n\_subframe - 1] \right| > 6\ dB.$$

3.5.9.3.3.7 CNG excitation generation

The CNG excitation signal *excitation*[] is computed from a multi-pulse excitation signal and a random excitation signal as follows:

```
for (i = 0; i < sbfrm_size; i++)
{
    excitation[i] = gf * fcb_cng[i] + gr * excg[i];
}
```

where fcb_cng[] and excg[] are respectively the multi-pulse (MP) or regular pulse (RP) excitation signal and a random excitation signal. gf and gr are their corresponding gains. The random excitation signal, positions and signs of pulses for the MP/RP excitation are sequentially produced from a random sequence in each subframe. In order to synchronise the random number CNG generator in the encoder and the decoder, the random excitation signal excg[] for a given subframe must be computed prior to the MP/RP Excitation generation fcb_cng[] for that subframe.

3.5.9.3.3.7.1 Random sequence

Random sequence are generated by the following function and are used for generation of the multi-pulse and the random excitation signals:

```
short Random (*seed)
{
    *seed = (short) ((int)(*seed * 31821 + 13849));
    return (*seed);
}
```

with an initial seed value of 21845 and commonly used for both excitations. This generator has a periodic cycle of 16 bits. The seed is initialized with 21845 at the beginning of every non-active period.

3.5.9.3.3.7.2 Generation of the random excitation

The random excitation signal of each subframe is a Gaussian random sequence, which is generated as follows:

```
for (i = 0; i < sbfrm_size; i++)
{
    excg[i] = Gauss (seed);
}
```

where

```
float Gauss (short *seed)
{
    temp = 0;
    for (i = 0; i < 12; i++)
    {
        temp += (float) Random (seed);
    }
    temp /= (2 * 32768);
    return (temp);
}
```

3.5.9.3.3.7.3 Multi-pulse excitation generation

In case MPE is used for coding of the voice-active frames, a multi-pulse excitation signal is generated for each subframe by randomly selecting the positions and signs of pulses. Multi-pulse structures of MPEG-4 version 1 CELP with MPE_Configuration = 24 and 31 are used for the sampling rate of 8 and 16 kbit/s, respectively. Positions and signs of 10 pulses are generated in a 40-sample vector. For subframe size of 80 samples, it is carried out twice to generate 20 pulses in an 80-sample vector. Indices of the positions and signs, mp_pos_idx and mp_sign_idx, are generated in each subframe as follows:

```
if (subframe size is 40 samples)
{
    setRandomBits (&mp_pos_idx, 20, seed);
    setRandomBits (&mp_sgn_idx, 10, seed);
}

if (subframe size is 80 samples)
{
    setRandomBits (&mp_pos_idx_1st_half, 20, seed);
    setRandomBits (&mp_sgn_idx_1st_half, 10, seed);
    setRandomBits (&mp_pos_idx_2nd_half, 20, seed);
    setRandomBits (&mp_sgn_idx_2nd_half, 10, seed);
}
```

where mp_pos_idx_1st_half and mp_sgn_idx_1st_half are indices of the positions and signs of the first half of the subframe and mp_pos_idx_2nd_half and 20 mp_sgn_idx_2nd_half are indices for the second half. The function setRandomBits() is defined in subclause 3.5.9.3.3.7.5.

3.5.9.3.3.7.4 Regular Pulse Excitation

In case RPE is used for coding of the voice-active frames, a regular pulse excitation signal is generated for each subframe. In case of non-active frames, the content of the adaptive codebook is initialised with zero. For the non-active frames only the fixed codebook is used. The fixed codebook excitation signal is generated by using a random shape_index as input for the RPE decoding process.

```
setRandomBits (&shape_index, n_bits, seed);

rpe_index = shape_index;
rpe_phase = rpe_index % D;
rpe_index = rpe_index / D;

for (n = Np - 1; n >= 0; n--)
{
   rpe_amps [n] = (rpe_index % 3) - 1;
   rpe_index = rpe_index / 3;
}

for (n = 0; n < sbfrm_size; n++)
{
   fcb_cng[n] = 0.0F;
}

for (n = 0; n < Np; n++)
{
   fcb_cng[rpe_phase + D*n] = gn[RPE_Configuration] * (float)(rpe_amps [n]);
}
```

nbits is set to 11 for RPE_Configurations 0 and 1 and set to 12 for RPE_configurations 2 and 3. D is the decimation factor, Np is the number of pulses per subframe and sbfrm_size is the number of samples per subframe as defined in MPEG-4 CELP version 1. The normalising factor gn is defined in Table 3.106.

Table 3.106 — Normalizing factor gn for RPE

RPE_Configuration	gn [ ]
0	56756 / 32768
1	56756 / 32738
2	44869 / 32768
3	40132 / 32768

3.5.9.3.3.7.5 Random index generator function

A random index generator function setRandomBits() is defined for MPE and RPE as follows:

```
void setRandomBits (long *l, int n, short *seed)
{
   *l = 0xffff & Random(seed);
   if (n > 16)
   {
      *l |= (0xffff & Random(seed)) << 16;
   }

   if (n < 32)
   {
      *l &= ((unsigned long)1 << n) - 1;
   }
```

```
}
```

3.5.9.3.3.7.6 Gain calculation

Gains *gf* and *gr* are calculated from the smoothed RMS of the excitation, *norm_sm* as follows:

$$gf = \alpha \cdot norm\_sm \Big/ \sqrt{\sum_{i=0}^{sbfrm\_size-1} fcb\_cng(i)^2 \Big/ sbfrm\_size}\ .$$

$$gr = [-\alpha \cdot A_3 + \sqrt{\alpha^2 \cdot A_3^2 - (\alpha^2 - 1) A_1 A_2}\ ] \Big/ A_2$$

where $\alpha = 0.6$ and

$$A_1 = \sum_{i=0}^{sbfrm\_size-1} gf^2\, fcb\_cng[i]^2$$

$$A_2 = \sum_{i=0}^{sbfrm\_size-1} excg[i]^2$$

$$A_3 = \sum_{i=0}^{sbfrm\_size-1} gf \cdot fcb\_cng[i] \cdot excg[i]\ .$$

3.5.9.3.3.8 LP Synthesis filter

The synthesis filter is identical to LP synthesis filter in MPEG-4 CELP described in subclause 3.5.8.

3.5.9.3.3.9 Memory update

Since the encoder and decoder need to be kept synchronized during non-active periods, the excitation generation is performed on both encoder and decoder sides to update the corresponding buffers for the LP synthesis. The adaptive codebook is not used and is initialized with zero during non-active frames.

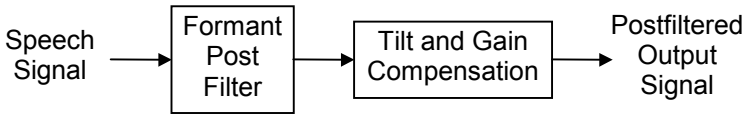
Annex 3.A
(informative)

MPEG-4 CELP decoder tools

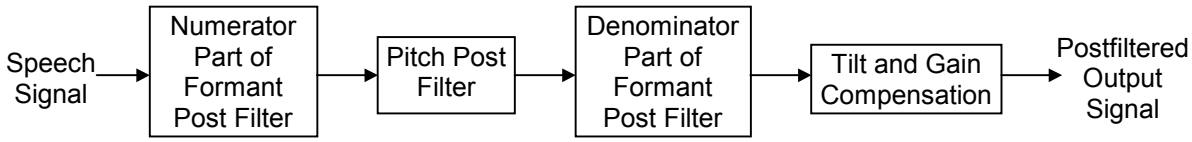
3.A.1 CELP post-processor

3.A.1.1 Tool description

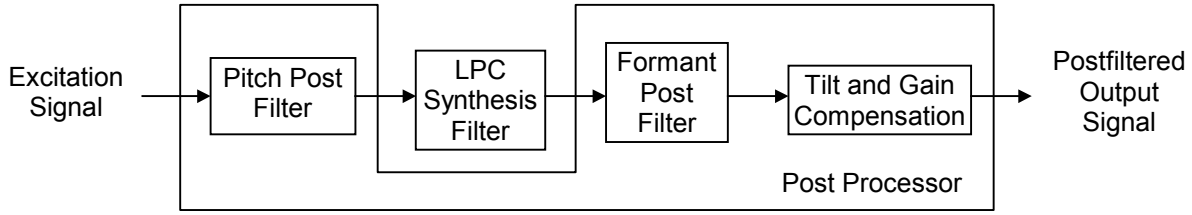
The CELP post-processor enhances the reconstructed speech signal generated by the synthesis filter for one subframe. The post-filtering tools comprise a formant post-filter and a tilt compensation post-filter. Some examples on implementation of the post-processor are shown in Figure 3.A.1.



(a) Post-processor with Formant Enhancement



(b) Post-processor with Formant and Pitch Enhancement



(c) Another Type of Post-processor with Formant and Pitch Enhancement

Figure 3.A.1 — Examples of post-processor

3.A.1.2 Definitions

Input

synth_signal[]: This array contains the reconstructed speech signal.

int_Qlpc_coefficients[]: This array contains the LPC coefficients for each subframe.

acb_delay: This field indicates the pitch delay, which is used for the pitch post-filter. If the pitch post-filter is not needed, *acb_delay* must be set to a value less than 10.

adaptive_gain: This field indicates the gain coefficient for the periodic component of the excitation signal. This gain coefficient is used for the pitch post-filter. If the pitch post-filter is not needed, *adaptive_gain* must be set to a value less than 0.4.

Output

PP_synth_signal[]: This array contains the post-filtered (enhanced) speech signal. The dimension of this array is sbfrm_size.

Configuration

lpc_order: This field indicates the order of LPC that is used.

sbfrm_size: This field indicates the number of samples in a subframe.

3.A.1.3 Decoding process

The decoding procedure is composed of post-filtering and adaptive gain control. The post-filter $H_{fpt}(z)$ is the cascade of three filters: a formant post-filter $H_f(z)$, a optional pitch post-filter $H_p(z)$, and a tilt compensation filter $H_t(z)$:

$$H_{fpt}(z) = H_f(z) \cdot H_p(z) \cdot H_t(z)$$

The formant post-filter is given by

$$H_f(z) = \frac{\widehat{A}(z/\gamma_n)}{\widehat{A}(z/\gamma_d)} = \frac{1 - \sum_{i=1}^{lpc\_order} \gamma_n^i \widehat{a}_i z^{-i}}{1 - \sum_{i=1}^{lpc\_order} \gamma_d^i \widehat{a}' z^{-i}},$$

where $\widehat{A}(z)$ is the LPC inverse filter and the factors γ_n and γ_d control the degree of formant post-filtering. γ_n and γ_d are set to 0.65 and 0.75, respectively.

The pitch post-filter is given by

$$H_p(z) = \frac{1}{1+\gamma_p g_a}(1 + \gamma_p g_a z^{-acb\_delay}),$$

$$g_a = \frac{\sum_{n=0}^{sbfrm\_size-1} s_r(n)s_r(n-acb\_delay)}{\sum_{n=0}^{sbfrm\_size-1} s_r^2(n-acb\_delay)},$$

where $s_r(n)$ is the residual signal produced by filtering the input signal through the numerator part of the formant post-filter. The gain coefficient g_a is bounded by 1. The factor γ_p controls the degree of pitch post-filtering and has the value of 0.5. The pitch post-filter is applied only if the gain is more than 0.4 and the pitch delay is more than 10. In the 16 kHz sampling rate for the Mode I coder, the pitch post-filter is not applied.

The filter $H_t(z)$ compensates the high-frequency tilt and is given by

$$H_t(z) = 1 - \gamma_t z^{-1},$$

where γ_t is a tilt factor of 0.3.

The synthesized signal (*synth_signal[]*) is filtered through the numerator part of the formant post-filter $\hat{A}(z/\gamma_n)$ to produce the residual signal. In the Mode II coder, the residual signal is then filtered through the pitch post-filter $H_p(z)$, whereas in the Mode I coder no pitch post-filter is used. The pitch post-filtered residual is fed through the denominator part of the formant post-filter $\hat{A}(z/\gamma_d)$. The output signal of the formant and pitch post-filters is passed through the tilt compensation filter $H_t(z)$ to generate the post-filtered synthesized signal that is not yet gain-compensated.

Adaptive gain control compensates gain differences between the synthesized signal $s(n)$ and the post-filtered signal $s_p(n)$. The gain control factor G for the current subframe is computed by

$$G = \sqrt{\frac{\sum_{n=0}^{sbfrm\_size-1} s^2(n)}{\sum_{n=0}^{sbfrm\_size-1} s_p^2(n)}}.$$

The gain-scaled post-filtered signal $s_p'(n)$ is given by

$$s'(n) = g(n)s_p(n), \quad 0 \le n \le sbfrm\_size - 1,$$

where $g(n)$ is updated on a sample-by-sample basis and given by

$$g(n) = 0.95g(n-1) + 0.05G, \quad 0 \le n \le sbfrm\_size - 1.$$

The initial value of $g(-1) = 0.0$ is used. Then for each new subframe, $g(-1)$ is set equal to $g(N-1)$ of the previous subframe.

Annex 3.B
(informative)

MPEG-4 CELP encoder tools

3.B.1 General Introduction to the MPEG-4 CELP encoder tool-set

This Annex provides a brief description of the functionality, parameter definition and the encoding processes of the tools supported by the MPEG-4 CELP core. The description of each tool comprises up to four parts: tool description, definitions, encoding process and tables.

The following encoder tools are supported:

- CELP preprocessing
- CELP LPC analysis
- CELP LPC quantizer and interpolator
  - Vector Quantizer
  - Bandwidth Scalable encoder
- CELP LPC analysis filter
- CELP weighting module
- CELP excitation analysis
  - regular pulse excitation
  - multi-pulse excitation
- CELP bitstream multiplexer

The encoding is performed on frame basis and each frame is divided in subframes. The CELP excitation analysis tool is used every subframe, while the other tools are used every frame.

3.B.2 Helping variables

For each encoder tool a description of the variables it uses is given. In this subclause the most commonly used variables are provided, which are shared by multiple tools.

frame_size: This field indicates the number of samples in a frame. The decoder outputs a frame with *frame_size* samples.

nrof_subframes: A frame is built up of a number of subframes. The number of subframes is specified in this field.

sbfrm_size: A subframe consists of a number of samples, which is indicated by this field. The number of samples in a frame must always be equal to the sum of the number of samples in the subframes. So, the following relation must always hold

frame_size = *nrof_subframes* * *sbfrm_size*

These three parameters depend on the sampling rate and the bitrate settings as tabulated in Table 3.73, for the Mode I coder and Table 3.74, for the Mode II coder.

lpc_order: This field indicates the number of coefficients used for Linear Prediction. By default, the value of this field is 20 for a sampling rate of 16 kHz and 10 for 8 kHz.

num_lpc_indices. This parameter specifies the number of indices containing LPC information, that must be written to the bitstream. This is not equal to the LPC-order. The *num_lpc_indices* is 5 in the 8 kHz mode and an additional 6 for the band scalable layer.

n_lpc_analysis: This field indicates how often LPC analysis is performed per frame. It is possible to perform several LPC analyses per frame with a varying window size and offset. For 16 kHz sampling rate, the value of this field is 1, indicating that LPC analysis is only performed once. For 8 kHz sampling rate, the value of this field is determined by the ratio *sbfrm_size*/80.

window_offsets[]: This array contains the offsets of the LPC analysis windows and its dimension is n_lpc_analysis.

window_sizes[]: This array contains the window sizes for the LPC analysis. Since the LPC analysis is performed n_lpc_analysis times, the dimension of this array is n_lpc_analysis.

Table 3.B.1 — Window size and offset for the 16 kHz mode I coder

RPE_Configuration	Window_sizes[] (samples)	Window_offsets[] (samples)
0	400	280
1	320	160
2	400	280
3	400	280
4....7	Reserved	

Table 3.B.2 — Window size and offset for the 8 kHz mode II coder

MPE_Configuration	Window_sizes[] (samples)	Window_offsets[] (samples)
0,1,2	200	0, 80, 160, 240
3,4,5	200	0, 80, 160
6 ... 12	200	0, 80
13 ... 21	200	0, 80
22 ... 26	200	0

Table 3.B.3 — Window size and offset for the 16 kHz mode II coder

MPE_Configuration	Window_sizes[] (samples)	Window_offsets[] (samples)
0 ... 6, 8 ... 15	320	0, 80, 160, 240
16 ... 22, 24 ... 31	320	0, 80

windows[]: This array contains the window for each analysis, so the length of this array is the sum of *window_sizes* times *n_lpc_analysis*. For the Mode I coder a squared Hamming window is used:

```
for (x = 0; x < window_sizes[i]; x++)
{
   window[i][x] = (0.54 - 0.46 * cos(2 * pi * (x/window_sizes[i])));
   window[i][x] = window[i][x] * window[i][x];
}
```

For the Mode II coder, a hybrid window is used. The window consists of two parts; the half of a Hamming window and a quarter of the cosine function, given by:

```
for (n = 0; n < nlb+len_lpcana/2; n++)
```

```
{
   Hw(n) = 0.54 - 0.46 * cos(2 * PI * n / (2*(nlb+len_lpcana/2) - 1));
}

for (n = nlb+len_lpcana/2; n < window_sizes[i]; n++)
{
   Hw(n) = cos (2 * PI * (n - (nlb+len_lpcana/2)) / (4*(nla+len_lpcana/2)-1));
}
```

nlb samples of the past analysis frame, *len_lpcana* samples of the present analysis frame, and *nla* samples of the future frame form one block for windowing.

Table 3.B.4 — Window parameters for the mode II coder

Sampling rate (kHz)	*nla*	*nlb*	*Len_lpc_ana*
8	40	80	80
16	80	160	80

gamma_be[]: This array is of size *lpc_order* and contains the weights for applying bandwidth expansion on the LPC coefficients. This information is used only for the mode I coder.

```
gamma_be[0] = GAMMA;
for (x = 1; x < lpc_order; x++)
{
   gamma_be[x] = GAMMA * gamma_be[x-1];
}
```

The value of GAMMA is 0.9883 for the RPE tool.

n_lag_candidates: This fields contains the number of pitch candidates. This information is used only for 16 kHz sampling rate, the value of this field is 15.

max_pitch_frequency: This field contains the maximum frequency of the pitch (lag). For 16 kHz sampling rate, this field has the value 0.025, since the minimum lag is 40. For 8 kHz sampling rate, this field has the value 0.05882, since the minimum lag is 17.

min_pitch_frequency: This field contains the minimum frequency of the pitch (lag). The value of this field for 16 kHz sampling rate is 3.3898E-3, since the maximum lag is 295. For 8 kHz sampling rate, this field has the value 6.944E-3, since the maximum lag is 144.

3.B.3 Bitstream elements for the MPEG-4 CELP encoder tool-set

See subclause 3.5.4.

3.B.4 CELP preprocessing

3.B.4.1 Tool description

The CELP preprocessing tool produces a DC free speech signal.

3.B.4.2 Definitions

Input

s[]: This is an array of dimension *frame_size* and contains input speech samples.

Output

pp_s []: This is an array of length *frame_size* and contains the DC free speech samples.

Input/Output

prev_x, prev_y: memory of the preprocessing filter

Configuration

*frame_size***:** This field indicates the number of samples in the input signal.

3.B.4.3 Encoding process

This block removes the DC element from the input signal, s[n]. It is first order recursive filter of the form:

$$H_{pre}(z) = \frac{1 - z^{-1}}{1 - 0.99 \cdot z^{-1}}$$

An implementation of this filter is given below:

```
for (n = 0; n < frame_size; n++)
{
   pp_s[n] = s[n] – prev_x + 0.99 * prev_y;
   prev_x = s[n];
   prev_y = pp_s[n];
}
```

The input/output filter states *prev_x* and *prev_y* are initialized to zero.

3.B.5 CELP LPC analysis

3.B.5.1 Tool description

The CELP LPC Analysis tool estimates the short-term spectrum. The LPC analysis is performed on the preprocessed speech signal, pp_s[]. The order of linear prediction is specified by the parameter lpc_order. A window, with size specified in window_size[], is used to weight the preprocessed speech. The parameter, window_offset[], is provided to specify the offset for each window.

3.B.5.2 Definitions

Input

PP_InputSignal[]: This array contains the preprocessed speech signal. This dimension is *frame_size*.

Output

lpc_coefficients[] : This array contains the computed LPC coefficients and has size *lpc_order*.

first_order_lpc_par : This output field contains the LPC coefficient for 1st-order fit. This parameter is used for the preselection in the adaptive codebook search.

Configuration

frame_size: This field denotes the number of samples in frame.

window_offset[]: This array contains the offset of the window.

window_size[]: This array contains the LPC analysis-window size.

windows[]: This array contains the windows used to weight the speech signal.

gamma_be[]: This array contains the gamma coefficients that are used for bandwidth expansion of the LPC coefficients.

lpc_order: This field indicates the order of LPC.

n_lpc_analysis: This field denotes the number of LPC analysis.

3.B.5.3 Encoding process

The Linear Prediction analysis is performed n_lpc_analysis times, each with a different window size and offset as specified in the array's window_size and window_offset. Each time the input signal PP_InputSignal is weighted, sw[n].

By means of the weighted signal the autocorrelation coefficients are derived using:

$$acf[k] = \sum_{n=0}^{window\_size-k-1} sw[n] \cdot sw[n+k], 0 \le k \le lpc\_order$$

There are *lpc_order*+1 autocorrelation coefficients. For the Mode II coder, a bandwidth expansion and a white-noise correction is applied by modifing the autocorrelation coefficients as follows:

```
for (k = 0; k < lpc_order; k++)
{
   acf[k] *= lag_win[k];
}
```

where *lag_win[]* is the coefficients for the bandwidth eapansion and is shown in Annex 3.D.

The LPC coefficients are computed using the conventional Levinson-Durbin recursion. The first LPC coefficient is assigned to first_order_lpc_par. For Mode I coder, bandwidth expansion is applied on the LPC coefficients using array gamma_be. For each LPC analysis, the calculated lpc_coefficients are stacked, resulting in *n_lpc_analysis* * *lpc_order* coefficients.

3.B.6 CELP LPC quantizer and interpolator

The LPC coefficients are quantized by using one of three quantizers, Narrowband LSP Quantization tool, Wideband LSP Quantization tool or Bandwidth Scalable LSP Quantization tool.

3.B.6.1 Narrowband LSP quantization tool

3.B.6.1.1 Tool description

The Narrowband LSP Quantization tool quantizes the LPC coefficients as LSP parameters using a two-stage and split vector quantization technique.

3.B.6.1.2 Definitions

Input

lpc_coefficients[]: This is an array of dimension *lpc_order* and contains the current unquantized LPC coefficients.

Output

int_Qlpc_coefficients[]: This is an array of length *nrof_subframes* * *lpc_order* and contains the interpolated and quantized LPC coefficients for each subframe. The LPC coefficients for each subframe are stacked one after the other, resulting in an *nrof_subframes* * *lpc_order* array.

lpc_indices[]: This is an array of dimension *num_lpc_indices* and contains the packed lpc indices that are written to the bitstream.

Configuration

lpc_order: This field contains the order of LPC.

num_lpc_indices: This fields indicates the number of packed LPC codes.

n_lpc_analysis: This field contains the number of LPC paramters.

nrof_subframes: This field contains the number of subframes.

3.B.6.1.3 Encoding process

The LPCs are converted into LSPs and quantized. As described in the decoding process, there are two methods for quantizing the LSPs; a two-stage VQ without interframe prediction, and a combination of the VQ and the interframe predictive VQ. At the encoding process, both methods are tried to quantize the LSPs and which method should be applied is determined by comparing the quantization error. The quantization error is calculated as a weighted Euclidean distance. The weighting coefficients (*w[]*) are,

$$w[\mathrm{i}] = \begin{cases} \dfrac{1}{lsp[0]} + \dfrac{1}{lsp[1] - lsp[0]} & (i = 0) \\ \dfrac{1}{lsp[i] - lsp[i-1]} + \dfrac{1}{lsp[i+1] - lsp[i]} & (0 < i < Np - 1) \\ \dfrac{1}{lsp[Np-1] - lsp[Np-2]} + \dfrac{1}{1.0 - lsp[Np-1]} & (i = Np - 1) \end{cases},$$

where *Np* is the LP analysis order *lpc_order* and *lsp[]* is the LSPs converted from the LPCs.

The first stage quantizer is the same for each quantization method. The LSPs are quantized by using a two-split vector quantizer and corresponding indices are stored in **lpc_indices[0]** and **lpc_indices[1]**. In order to carry out delayed decision, two indices, namely the best and the second best, are stored as candidates for the second stage. The quantization error in the first stage *err1*[] is given by:

$$err1[n] = \sum_{i=0}^{\dim-1} \left\{ \left(lsp[sp+i] - lsp\_tbl[n][m][i] \right)^2 \cdot w[sp+i] \right\} \qquad n = 0,1$$

where *n* is the split vector number, *m* is the index of the candidate split vector , sp is the starting LSP order of the n-th split vector and dim is the dimension of the n-th split vector. (*lsp_tbl[][][]* is shown in Annex 3.C)

Table 3.B.5 — Starting order and dimension of the first stage LSP vector

Split vector number: n	Starting LSP order: sp	Dimension of the vector: dim
0	0	5
1	5	5

In the second stage, the above-mentioned two quantization methods, which are also two-split vector quantizers, are applied respectively. Total quantization errors in the second stage are calculated for all combinations of the first stage candidates and the second stage candidates and the one that has the minimum error is selected. As a result, indices of the first stage are determined and corresponding indices and signs for the second stage are stored in **lpc_indices[2]** and **lpc_indices[3]**. The flag that indicates the selected quantization method is also stored in **lpc_indices[4]**. The quantization error in the second stage *err2_total* is given by:

VQ without interframe prediction:

$$err2\_total = err2[0] + err2[1]$$

$$err2[n] = \sum_{i=0}^{\dim-1} \left\{ \left(lsp\_res[sp+i] - sign[n] \cdot d\_tbl[n][m][i] \right)^2 \cdot w[sp+i] \right\} \qquad n = 0,1$$

$$lsp\_res[sp+i] = lsp[sp+i] - lsp\_first[sp+i]$$

where *lsp_first*[] is the quantized LSP vector of the first stage, *n* is the split vector number, *m* is the index of the candidate split vector , sp is the starting LSP order of the n-th split vector and dim is the dimension of the n-th split vector. (*d_tbl[][][]* is shown in Annex 3.C)

VQ with interframe prediction:

$$err2\_total = err2[0] + err2[1]$$

$$err2[n] = \sum_{i=0}^{\dim-1} \left\{ \left(lsp\_pres[sp+i] - sign[n] \cdot pd\_tbl[n][m][i] \right)^2 \cdot w[sp+i] \right\} \qquad n = 0,1$$

$$lsp\_pres[sp+i] = lsp[sp+i] - \left\{ \left(1 - ratio\_predict\right) \cdot lsp\_first[sp+i] + ratio\_predict \cdot lsp\_previous[sp+i] \right\}$$

where *lsp_first*[] is the quantized LSP vector of the first stage, *n* is the split vector number, *m* is the index of the candidate split vector , sp is the starting LSP order of the n-th split vector, dim is the dimension of the n-th split vector and ratio_predict=0.5. (*pd_tbl[][][]* is shown in Annex 3.C)

Table 3.B.6 — Starting order and dimension of the second stage LSP vector

Split vector number: n	Starting LSP order: sp	Dimension of the vector: dim
0	0	5
1	5	5

The quantized LSPs *lsp_current[]* are stabilized in order to ensure stability of the LPC synthesis filter, which is derived from the quantized LSPs. The quantized LSPs are arranged in order, having a distance of at least *min_gap* between adjacent coefficients.

```
for (i = 0; i < lpc_order; i++)
{
   if (lsp_current[i] < min_gap)
   {
      lsp_current[i] = min_gap;
   }
}


for (i = 0; i < lpc_order - 1; i++)
{
   if (lsp_current[i+1] - lsp_current[i] < min_gap)
   {
      lsp_current[i+1] = lsp_current[i]+min_gap;
   }
}


for (i = 0; i < lpc_order; i++)
{
   if (lsp_current[i] > 1-min_gap)
   {
      lsp_current[i] = 1-min_gap;
   }
}


for (i = lpc_order - 1; i > 0; i--)
{
   if (lsp_current[i]-lsp_current[i-1] < min_gap)
   {
      lsp_current[i-1] = lsp_current[i]-min_gap;
   }
}
```

where min_gap = 2.0/256.0

After the quantization process, the quantized LSPs are interpolated linearly at each subframe.

```
for (n = 0; n < nrof_subframes; n++)
{
   ratio_sub=(n+1)/nrof_subframes;
   for (i = 0; i < lpc_order; i++)
   {
      lsp_subframe[n][i]=((1-ratio_sub)*lsp_previous[i] + ratio_sub*lsp_current[i]));
   }
}
```

The interpolated LSPs are converted to the LPCs using the auxiliary function *Convert2lpc().*

```
for (n = 0; n < nrof_subframes; n++)
{
   Convert2lpc(lpc_order, lsp_subframe[n], int_Qlpc_coefficients + n*lpc_order]);
}
```

After calculation of the LPC coefficients, the current LSPs must be stored in memory, since they are used for interpolation at the next frame.

```
for (i = 0; i < lpc_order; i++)
{
   lsp_previous[i] = lsp_current[i];
}
```

It should be noted that the stored LSPs *lsp_previous[]* must be initialized as described below when the entire encoder is initialized.

```
for (i = 0; i < lpc_order; i++)
{
   lsp_previous[i] = (i+1) / (lpc_order+1);
}
```

3.B.6.2 Wideband LSP Quantization Tool

3.B.6.2.1 Tool description

The Wideband LSP Quantization tool quantizes the LPC coefficients as LSP parameters using a two-stage and split vector quantization technique.

3.B.6.2.2 Definitions

Input

lpc_coefficients[]: This is an array of dimension *lpc_order* and contains the current unquantized LPC coefficients.

Output

int_Qlpc_coefficients[]: This is an array of length *nrof_subframes* * *lpc_order* and contains the interpolated and quantized LPC coefficients for each subframe. The LPC coefficients for each subframe are stacked one after the other, resulting in a *nrof_subframes* * *lpc_order* array.

lpc_indices[]: This is an array of dimension *num_lpc_indices* and contains the packed lpc indices that are written to the bitstream.

Configuration

lpc_order: This field contains the order of LPC.

num_lpc_indices: This fields indicates the number of packed LPC codes.

n_lpc_analysis: This field contains the number of LPC parameters.

nrof_subframes: This field contains the number of subframes.

3.B.6.2.3 Encoding process

The quantization scheme is based on the Narrowband LSP Quantization. The quantizer consists of two quantization blocks connected in parallel, each of which is identical to the Narrowband LSP Quantization tool. The input LSPs are divided into two parts, namely the lower part and the upper part, then the divided LSPs are input to the quantization blocks respectively.

First, the lower part is quantized in the same manner as the Narrowband LSP Quantization. The starting order and the dimension of the LSP vectors are described in Table 3.B.7 and

Table 3.B.8. The quantized LSPs are stored in the array *lsp_current_lower[].*

Table 3.B.7 — Starting order and dimension of the first stage LSP vector

Split vector number: n	Starting order of the lower LSP order: sp	Dimension of the vector: dim
0	0	5
1	5	5

Table 3.B.8 — Starting order and dimension of the second stage LSP vector

Split vector number: n	Starting order of the lower LSP order: sp	Dimension of the vector: dim
0	0	5
1	5	5

Next, the upper part is quantized in the same manner. The starting order and the dimension of the LSP vectors are described in Table 3.B.9 and

Table 3.B.10. The quantized LSPs are stored in the array *lsp_current_upper[].* In the quantization of the upper part, indices of the first stage are stored in **lpc_indices[5]** and **lpc_indices[6]**, and indices and signs for the second stage are stored in **lpc_indices[7]** and **lpc_indices[8]**. The flag that indicates the selected quantization method is also stored in **lpc_indices[9]**.

Table 3.B.9 — Starting order and dimension of the first stage LSP vector

Split vector number: n	Starting order of the upper LSP order: sp	Dimension of the vector: dim
0	0	5
1	5	5

Table 3.B.10 — Starting order and dimension of the second stage LSP vector

Split vector number: n	Starting order of the upper LSP order: sp	Dimension of the vector: dim
0	0	5
1	5	5

Finally, the decoded LSPs *lsp_current_lower[]* and *lsp_current_upper[]* are combined and stored in the array *lsp_current[].*

```
for (i = 0; i < lpc_order/2; i++)
{
   lsp_current[i] = lsp_current_lower[i];
}
for (i = 0; i < lpc_order/2; i++)
{
   lsp_current[lpc_order/2+i] = lsp_current_upper[i];
}
```

The quantized LSPs *lsp_current[]* are stabilized in order to ensure stability of the LPC synthesis filter, which is derived from the quantized LSPs. The quantized LSPs are arranged in ascending order, having a distance of at least *min_gap* between adjacent coefficients.

```
for (i = 0; i < lpc_order; i++)
{
   if (lsp_current[i] < min_gap)
   {
      lsp_current[i] = min_gap;
   }
}

for (i = 0; i < lpc_order - 1; i++)
{
   if (lsp_current[i+1] - lsp_current[i] < min_gap)
   {
      lsp_current[i+1] = lsp_current[i]+min_gap;
   }
}

for(i = 0; i < lpc_order; i++)
{
   if (lsp_current[i] > 1-min_gap)
   {
      lsp_current[i] = 1-min_gap;
   }
}

for (i = lpc_order - 1; i > 0; i--)
{
   if (lsp_current[i]-lsp_current[i-1] < min_gap)
   {
      lsp_current[i-1] = lsp_current[i]-min_gap;
   }
}
```

where lpc_order = 20 and min_gap = 1.0/256.0

After the quantization process, the quantized LSPs are interpolated linearly at each subframe.

```
for (n = 0; n < nrof_subframes; n++)
{
   ratio_sub=(n+1)/nrof_subframes;
   for (i = 0; i < lpc_order; i++)
   {
      lsp_subframe[n][i]=((1-ratio_sub)*lsp_previous[i] + ratio_sub*lsp_current[i]));
   }
}
```

The interpolated LSPs are converted to the LPCs using the auxiliary function *Convert2lpc().*

```
for (n = 0; n < nrof_subframes; n++)
{
   Convert2lpc(lpc_order, lsp_subframe[n], int_Qlpc_coefficients + n*lpc_order]);
}
```

After calculation of the LPC coefficients, the current LSPs must be stored in memory, since they are used for interpolation at the next frame.

```
for (i = 0; i < lpc_order; i++)
{
   lsp_previous[i] = lsp_current[i];
}
```

It should be noted that the stored LSPs *lsp_previous[]* must be initialized as described below when the entire encoder is initialized.

```
for (i = 0; i < lpc_order; i++)
{
   lsp_previous[i] = (i+1) / (lpc_order+1);
}
```

3.B.6.3 Bandwidth scalable LSP quantization tool

3.B.6.3.1 Tool description

The Bandwidth Scalable LSP Quantization tool quantizes the input 16 kHz sampling rate LSPs using vector quantization scheme with intraframe and interframe prediction.

3.B.6.3.2 Definitions

Input

lpc_coefficients[]: This is an array of dimension *lpc_order* and contains the current unquantized LPC coefficients.

lsp_current[]: This array contains the decoded LSP parameters, which are normalized in the range of zero to PI, at the Narrowband LSP Quantization tool. These parameters are obtained as intermediate parameters in the Narrowband LSP Quantization process and forwarded to the Bandwidth Scalable LSP Decoding tool.

Output

int_Qlpc_coefficients[]: This is an array of length *nrof_subframes* * *lpc_order* and contains the interpolated and quantized LPC coefficients for each subframe. The LPC coefficients for each subframe are stacked one after the other, resulting in an *nrof_subframes* * *lpc_order* array.

lpc_indices[]: This is an array of dimension *num_lpc_indices* and contains the packed lpc indices that are written to the bitstream.

Configuration

lpc_order: This field contains the order of LPC.

num_lpc_indices: This fields indicates the number of packed LPC codes.

n_lpc_analysis: This field contains the number of LPC parameters.

nrof_subframes: This field contains the number of subframes.

nrof_subframes_bws: This parameter, which is a help variable, represents the number of subframes in the bandwidth scalable layer.

3.B.6.3.3 Encoding process

The input 16 kHz sampling rate LSPs (*input_lsp[]*) are vector-quantized with intraframe and interframe prediction approaches. The intraframe prediction module produces estimated LSPs by converting the quantized LSPs obtained in the 8 kHz sampling rate CELP coder. Furthermore, an interframe moving average predictive VQ is also employed for more accurate prediction. The prediction residual LSPs (*err_lsp[]*) are computed as follows:

```
for (i = 0; i < 20; i++)
{
   err_lsp[i] = (input_lsp[i] - pred_lsp[i]) / bws_ma_prdct[0][i];
}

for (n = 1; n <= 2; n++)
{
   for (i = 0; i < 20; i++)
   {
      pred_lsp[i] += bws_ma_prdct[n][i]*lsp_bws_buf[n][i];
   }
}

for (i = 0; i < 10; i++)
{
   pred_lsp[i] += bws_nw_prdct[i]*lsp_current[i];
}
```

where *pred_lsp[]* contains the predicted LSPs. *bws_ma_prdct[][]* and *bws_nw_prdct[]* are prediction coefficients for moving average interframe prediction and intraframe prediction, respectively. *lsp_bws_buf[][]* is a buffer containing LSP prediction residual at the previous two frames.

Then, *err_lsp[]* are vector-quantized using a two-stage and split vector quantization scheme and corresponding indices are stored in **lpc_indices[5], ... , lpc_indices[10].** The buffer *lsp_bws_buf[][]* is shifted for the next frame operation as follows:

```
for (i = 0; i < 20; i++)
{
   lsp_bws_buf[0][i] = err_lsp[i];
}

for (n = 2; n > 0; n--)
{
   for (i = 0; i < 20; i++)
```

```
    {
       lsp_bws_buf[n][i] = lsp_bws_buf[n-1][i];
    }
 }
```

After the quantization process, the quantized LSPs (*lsp_bws_current[]*) are interpolated linearly at each subframe.

```
for (n = 0; n < nrof_subframes_bws; n++)
{
   ratio_sub = (n+1)/nrof_subframes_bws;
   for(i = 0; i < 2*lpc_order; i++)
   {
      lsp_bws_subframe[n][i]=((1-ratio_sub)*lsp_bws_previous[i]
                              + ratio_sub*lsp_bws_current[i]));
   }
}

for (i = 0; i < 2*lpc_order; i++)
{
   lsp_bws_previous[i] = lsp_bws_subframe[nrof_subframes_bws-1][i];
}
```

The interpolated LSPs are converted to the LPC coefficients at each subframes.

```
for (n = 0; n < nrof_subframes_bws; n++)
{
   Convert2lpc (lpc_order_bws, lsp_bws_subframe[n],
                              &int_Qlpc_coefficients[n*lpc_order_bws]);
}
```

3.B.6.4 Fine rate control in the LSP quantization tool

3.B.6.4.1 Tool description

Fine Rate Control (FRC) is available with the LSP Quantization tool. If the FRC is used, the array *lpc_coefficients[]* contains the LPC coefficients of the next frame which is the frame following the frame currently processed. The decision is made whether or not the LSPs of the currently processed frame are transmitted to the decoder. According to the decision, the **interpolation_flag** and **LPC_present** are set to 1 or 0.

3.B.6.4.2 Definitions

Input

lpc_coefficients[]: This is an array of dimension *lpc_order* and contains the current unquantized LPC coefficients.

Output

int_Qlpc_coefficients[]: This is an array of length *nrof_subframes* * *lpc_order* and contains the interpolated and quantized LPC coefficients for each subframe. The LPC coefficients for each subframe are stacked one after the other, resulting in an *nrof_subframes* * *lpc_order* array.

lpc_indices[]: This is an array of dimension *num_lpc_indices* and contains the packed lpc indices that are written to the bitstream.

interpolation_flag: This field indicates if interpolation on the LPC coefficients between frames is done. If the interpolation flag is set, the current LPC coefficients are computed from the previous and next LPC coefficients.

LPC_Present: This flag indicates whether the current frame is complete or incomplete.

3.B.6.4.3 Encoding process

The decision whether or not the LPC coefficients of the speech frame under analysis are transmitted to the decoder depends on the amount of change *d*, between the spectrum of the current frame and the spectrum of the adjacent frames. If *d* is greater than a particular threshold then the coefficients are transmitted to the decoder. This threshold is further made dependent on the desired bitrate setting as follows: The threshold is raised if the actual bitrate is higher than the desired bitrate setting and is lowered otherwise.

3.B.7 CELP LPC analysis filter

3.B.7.1 Tool description

The CELP LPC Analysis filter tool feeds the input signals through a filter with LPC coefficients and returns the residual signal.

3.B.7.2 Definitions

Input

PP_InputSignal[]: This array has dimension sbfrm_size and contains the input signal.

int_Qlpc_coefficients[]: This array has dimension *lpc_order* and contains the LPC coefficients.

Output

lpc_residual[]: This array has dimension sbfrm_size and contains the LPC filtered residual signal.

Configuration

lpc_order: This field indicates the order of LPC.

sbfrm_size: This field indicates the number of samples in one subframe.

3.B.7.3 Encoding process

The input signal is filtered using the filter coefficients.

```
for (k = 0; k < sbfrm_size; k++)
{
   tmp = PP_InputSignal[k];
   for (j = 0; j < lpc_order; j++)
   {
      tmp =  tmp - int_Qlpc_coefficients[j] * Filter_States[j];
   }
   lpc_residual[k] = tmp;
   update_Filter_States;
}
```

The initial filter states are set to zero.

3.B.8 CELP weighting module

3.B.8.1 Tool description

The CELP weighting module computes the weights to be applied on the LPC coefficients.

3.B.8.2 Definitions

Input

lpc_coefficients[]: This is an array of dimension *lpc_order*, containing the LPC Coefficients.

gamma_num: This field contains the weighting factor of the numerator.

gamma_den: This field contains the weighting factor of the denominator.

Output

Wnum_coeff[]: This is an array of dimension Wnum_order, containing the weighted numerator coefficients.

Wden_coeff[]: This array has dimension Wden_order and contains the weighted denominator coefficients.

Configuration

lpc_order: This field contains the order of the LPC.

3.B.8.3 Encoding process

The weighted coefficients of the numerator are computed by:

```
for (k = 0; k < Wnum_order; k++)
{
   Wnum_coeff[k] = lpc_coefficients[k] * gamma_num^(k+1);
}
```

The weighted coefficients of the denominator are computed by:

```
for (k = 0; k < Wden_order; k++)
{
   Wden_coeff[k] = lpc_coefficients[k] * gamma_den^(k+1);
}
```

3.B.9 CELP excitation analysis

The CELP excitation analysis computes the shape and gain vectors as well as the decoded synthesized speech signal. For excitation analysis modules, Regular Pulse Excitation (RPE) and Multi Pulse Excitation (MPE) are defined.

3.B.9.1 Regular pulse excitation

3.B.9.1.1 Tool description

For Regular Pulse Excitation, the output of the excitation analysis are **shape_delay[]**, **shape_index[]** and **gain indices[]**. The shape and gain indices are generated every subframe. The vector **shape_delay[]** contains the adaptive codebook lag for each subframe, while the vector **shape_index[]** contains the fixed codebook index. The vector **gain_indices[0][]** contains the adaptive codebook gain for every subframe. The fixed codebook gain for every subframe is stored in vector **gain_indices[1][]**.

3.B.9.1.2 Definitions

Input

PP_inputSignal[]: This array contains the preprocessed input signal and is of dimension sfrm_size.

lpc_residual[]: This contains the lpc residual signal.

int_Qlpc_coefficients[]: This array contains the interpolated and quantized LPC coefficients.

Wnum_coeff[]: This array contains the weighting filter coefficients of the numerator.

Wden_coeff[]: This array contains the weighting filter coefficients of the denominator.

first_order_lpc_par: This field indicates the first LPC coefficient.

lag_candidates[]: This array contains the lag candidates.

signal_mode: This field contains the voiced/unvoiced flag .

rms_index: This field defines the index for the signal power.

Output

shape_delay[]: This array is of dimension *nrof_subframes*. It contains the codebook lag for the adaptive codebooks.

shape_index[]: This array is of dimension *nrof_subframes*. It contains the shape indices for the fixed codebooks.

gain_indices[][]: This array is of dimension 2 * nrof_subframes. It contains the gain indices for the adaptive and fixed codebooks.

decoded_excitation[]: This array is of length sbfrm_size and contains the synthesized speech signal.

Configuration

n_lag_candidates: This field indicates the number of lag candidates.

frame_size: This field indicates the number of samples in one frame.

sbfrm_size: This field indicates the number of samples in one subframe.

nrof_subframes: This field indicates the number of subframes in one frame.

lpc_order: This field indicates the order of LPC.

3.B.9.1.3 Encoding process

All blocks are performed on a once per subframe basis. For each subframe the following steps are performed:

- perceptual weighting
- adaptive codebook search preselection
- adaptive codebook search
- fixed codebook search preselection
- fixed codebook search
- simulation of decoder

Next, each step will be described in detail. For convenience, *aq*[] is used instead of *int_Qlpc_coefficients*[], so the *aq*[] contains the quantized LPC coefficients for the subframe under consideration.

Perceptual weighting is performed by filtering the input signal *PP_InputSignal[]* by the following filter:

$$W(z) = \frac{A(z)}{A(z/\gamma)} = \frac{1 - \sum_{k=0}^{lpc\_order-1} aq[k] \cdot z^{-k-1}}{1 - \sum_{k=0}^{lpc\_order-1} aq[k] \cdot \gamma^{k+1} \cdot z^{-k-1}}$$

with $\gamma = 0.8$. The resulting signal is called *ws[n].*

The zero-input response *z[n]*, is determined by computing the response of *S(z*) to an zero-valued input signal.

$$S(z) = \frac{1}{A(z/\gamma)} = \frac{1}{1 - \sum_{k=0}^{lpc\_order-1} aq[k] \cdot \gamma^{k+1} \cdot z^{-k-1}}$$

where $\gamma = 0.8$.

The weighted target signal *t[n]* is obtained by subtracting *z[n]* from *ws[n]*:

```
for (n = 0; n < Nm; n++)
{
   t[n] = ws[n] - z[n];
}
```

The impulse response *h[n]* is calculated as follows:

```
for (k = 0; k < lpc_order; k++)
{
   tmp_states[k] = 0;
}
tmp = 1.0;
for (n = 0; n < sbfrm_size; n++)
{
   g = 0.8;
   for (k = 0; k < lpc_order; k++)
   {
```

```
      tmp = tmp + aq[k][s] * g * tmp_states[k];
      g = 0.8 * g;
   }
   h[n] = tmp;
   for (k = lpc_order-1; k > 0; k--)
   {
      tmp_states[k] = tmp_states[k-1];
   }
   tmp_states[0] = tmp;
}
```

After the above mentioned computations, preselection on the adaptive codebook is performed. The adaptive codebook contains 256 codebook sequences of which 5 are preselected. Preselection is achieved by evaluating the term (for 0 <= l < Lm and 0 <= i < *nrof_subframes*):

$$rap[i \cdot Lm + l] = \frac{\left(\sum_{n=0}^{sbfrm\_size-1} ca[L\min + i \cdot sbfrm\_size + l \cdot 3 - n - 1] \cdot ta[n] \right)^2}{Ea[i \cdot Lm + l]}$$

where:

$$Lm = 1 + \left\lfloor \frac{sbfrm\_size - 1}{3} \right\rfloor$$

and *ca*[Lmin + *i* *sbfrm_size* + 3 *l* - *n*-1] and *ta*[*n*] representing the codebook sequence at delay (Lmin+*i* *sbfrm_size*+3 *l*) and the 'backward-filtered' target signal *t*[*n*], respectively.

Backward filtering comprises the time-reversal *of t[n],* filtering by *S(z)* and time-reversal again. *Lmin* is the minimum lag in samples. The value of *Lmin* is 40.

Ea[i Lm+l] is the energy of that codebook sequence filtered by a reduced complexity synthesis filter $S_p(z)$ which is a first-order estimation of the LPC synthesis filter *S* (*z*):

$$S_p(z) = \frac{1}{A_p(z/\gamma)} = \frac{1}{1 - a \cdot \gamma \cdot z^{-1}}$$

The first LPC coefficient of a frame is taken for *a*. Depending on the number of subframe, which is under consideration, the first LPC coefficient of the current or the previous is taken using the following:

```
if (subframe_number < nr_subframes/2)
{
   a = prev_a;
}
else
{
   a = cur_a;
}
```

After evaluation of *rap[]* the 5 maximum sequences are selected together with their 2 neighbor sequences resulting in 15 candidates on which full search is applied. The indices of the preselected sequences and their neighbors are stored in *ia*[*r*] (0 <= *r* < 15).

The adaptive codebook search minimizes the mean-square weighted error between the original and reconstructed speech. This is achieved by searching for the index *r* maximizing the term

$$ra[r] = \frac{\left(\sum_{n=0}^{sbfrm\_size-1} t[n] \cdot y[r][n]\right)^2}{\sum_{n=0}^{sbfrm\_size-1} y^2[r][n]}$$

The signal *t*[*n*] represents the weighted target signal and *y*[*r*][*n*] is the convolution of the sequence *ca*[*ia*[*r*]-*n*-1] with the impulse response *h*[*n*]. The index (*ia*[*r*]-*Lmin*) of the maximum is referred as **shape_delay[**subframe**]**.

After determining the index the gain factor is computed according to

$$g = \frac{\sum_{n=0}^{sbfrm\_size-1} t[n] \cdot y'[n]}{\sum_{n=0}^{sbfrm\_size-1} y'^2[n]}$$

with *y'*[*n*] equal to the convolution of *ca*[*I*+Lmin-*n*-1] with *h*[*n*]. The gain factor is quantized by a non-uniform quantizer:

```
for (j = 0; abs(g) > cba_gain_quant[j] && j < 31; j ++);

if (g < 0)
{
   Ga =  -cba_gain[j];
   gain_indices[0] = (((-j - 1)) & 63);
}
else
{
   Ga = cba_gain[j];
   Gain_indices[0][subframe] = j;
}
```

The quantized gain factor is referred as *Ga*. With **shape_delay[]** and *Ga* the contribution *p*[*n*] of the adaptive codebook is computed according to

$$p[n] = Ga \cdot y'[n]$$

The contribution *p*[*n*] of the adaptive codebook is subtracted from the weighted target signal *t*[*n*] to obtain the residual signal *e*[*n*]:

$$e[n] = t[n] - p[n], 0 \le n < sbfrm\_size$$

The residual signal *e*[*n*] is 'backward-filtered' to obtain *tf*[*n*].

The fixed codebook search also consists of a preselection and a selection phase. An RPE-codebook is used for the fixed codebook excitation. Each codebook vector has sbfrm_size pulses of which Np pulses may have an amplitude of +1, 0 or -1. These Np pulses are positioned on a regular grid defined by the phase *p* and the pulse spacing D such that the grid positions are at *p* + D*I* where *I* is between 0 and Np. The leaving (sbfrm_size-Np) pulses are zero. D and Np are dependent on the bitrate setting as is given in Table 3.87.

To reduce complexity a local RPE-codebook is generated for each subframe containing a subset of 16 entries. All vectors of this local RPE-codebook have the same phase P which computed as follows:

```
max = 0;
for (p = 0; p < D; p++)
{
   tmp_max = 0;
   for (l = 0; l < Np-1; l++)
   {
      tmp_max += abs(tf[p+D * l]);
   }
   if (tmp_max > max)
   {
      max = tmp_max;
      P = p;
   }
}
```

Having determined the phase P it is required that the amplitude of pulse l, 0 <= l < Np, is either zero or equal to the sign of the corresponding sample of tf[n]. The sign is stored in amp[l] as follows:

```
for (l = 0; l < Np; l++)
{
   amp[l] = sign (tf[P + D * l]);
}
```

where sign is +1 for values greater than 0 and –1 otherwise.

For Np-4 pulses the possibility of zero-amplitude is excluded a-priori at those positions where tf[n] is maximal. Therefore, array pos[] is used, which has the following semantic:

- pos[l] = 0 indicates that zero-amplitude possibility is included for pulse l.
- pos[l] = 1 indicates that zero-amplitude possibility is excluded.

The following algorithm is used to determine the array pos[]:

1. initialize pos[] with zeros,
2. Determine n such that abs(tf[P + D * n]) >= abs(tf[P + D * i]) for all i not equal to n
3. pos[n] = 1
4. tf[P + D * n] = 0
5. Proceed with step 2 until Np-4 positions are fixed

Based on P, amp[] and pos the local RPE-codebook cf[k][n] is generated, using the following algorithm:

```
for (k = 0; k < 16; k++)
{
   for (n = 0; n < sbfrm_size}; n++)
   {
      cf[k][n] = 0;
   }
}
for (l = 0; l < Np; l++)
{
   cf[0][P+D*l]=amp[l];
}

m = 1;
for (l = 0; l < Np; l++)
{
   if (pos[l]==0)
   {
      c = m;
      for (q = 0; q < c; q++)
      {
         for (n = 0; n < Np; n++)
         {
            cf[m][P+D*n] = cf[q][P+D*n];
         }
         cf[m++][P+D*l] = 0;
      }
   }
}
```

Based on the local RPE-codebook, 5 vectors out of 16 are preselected for the closed-loop search. Preselection is achieved by evaluating the term for 0 <= k < 16:

$$rfp[k] = \frac{\left(\sum_{n=0}^{sbfrm\_size-1} cf[k][n] \cdot tf[n]\right)^2}{Ef[l]}$$

with *cf*[*k*][*n*] representing a vector of the local RPE-codebook. *Ef*[*k*] is the energy of that codebook vector filtered by the reduced complexity synthesis filter $S_p(z)$. The indices of the preselected vectors are stored in *if*[*r*].

Using these preselected indices, a closed-loop fixed codebook search is performed. By the closed-loop fixed codebook search it is searched for the index *r* maximizing the term

$$rf[r] = 2 \cdot Gf \cdot \sum_{n=0}^{sbfrm\_size-1} e[n] \cdot y[r][n] - Gf^2 \cdot \sum_{n=0}^{sbfrm\_size-1} y^2[r][n]$$

The signal *e*[*n*] represents the residual signal of the adaptive codebook search and *y*[*r*][*n*] is the convolution of *cf*[*if*[*r*]][*n*] with the impulse response *h*[*n*]. *Gf* is the quantized gain factor *g* which is determined according to

$$g = \frac{\sum_{n=0}^{sbfrm\_size-1} e[n] \cdot y[r][n]}{\sum_{n=0}^{sbfrm\_size-1} y^2[r][n]}$$

The quantization process of the fixed codebook gain is given below, where *Gp* is the previous quantized fixed codebook gain.

```
if (first subframe)
{
   for (m = 0; g > cbf_gain_quant[m] && m < 30; m ++);
   Gf = cbf_gain[m];
}
else
{
   g = g / Gp;
   for (m = 0; g > cbf_gain_quant_dif[m]  && m < 7; m ++);
   g = Gp * cbf_gain_diff[m];
}
gain_indices[0] = m;
```

The representation tables cbf_gain and cbf_gain_dif are given in Table 3.89 and Table 3.90, whereas the quantization tables cbf_gain_quant and cbf_gain_quant_dif are given below.

Finally, the decoder is simulated by performing the following steps:

- the excitation of the fixed codebook is computed
- the excitation of the fixed and adaptive codebook are summed
- the memory of the adaptive codebook is updated
- the memory of the filters is updated

A detailed description of these steps is given in the decoder.

Tables used for gain quantization

Table 3.B.11 — Quantization table for adaptive codebook gain

Index	Cba_gain_quant	Index	cba_gain_quant
0	0.1622	16	0.9989
1	0.2542	17	1.0539
2	0.3285	18	1.1183
3	0.3900	19	1.1933
4	0.4457	20	1.2877
5	0.4952	21	1.4136
6	0.5425	22	1.5842
7	0.5887	23	1.8559
8	0.6341	24	2.3603
9	0.6783	25	3.8348
10	0.7227	26	7.6697
11	0.7664	27	15.339
12	0.8104	28	30.679
13	0.8556	29	61.357
14	0.9005	30	122.71
15	0.9487	31	245.43

Table 3.B.12 — Quantization table for fixed codebook gain

Index	Cbf_gain_quant	Index	cbf_gain_quant
0	2.4726	16	82.3374
1	3.1895	17	95.3755
2	4.2182	18	109.8997
3	5.6228	19	126.3037
4	7.3781	20	144.3995
5	9.5300	21	165.5142
6	12.1013	22	190.9742
7	15.2262	23	220.6299
8	19.0319	24	258.2699
9	23.6342	25	305.5086
10	29.1562	26	368.5894
11	35.3606	27	453.5156
12	42.8301	28	573.6164
13	51.1987	29	801.6422
14	60.6440	30	9999.9
15	70.9884	31	------

Table 3.B.13 — Quantization table for differential fixed codebook gain

Index	cbf_gain_quant_dif	Index	cbf_gain_quant_dif
0	0.2500	4	1.3356
1	0.5378	5	1.8869
2	0.7795	6	4.2000
3	1.0230	7	9999.9

3.B.9.2 Multi-pulse excitation

3.B.9.2.1 Tool description

For the CELP coder based on Multi Pulse Excitation, the output of the excitation analysis are **signal_mode, rms_index, shape_delay[]**, **shape_positions[]**, **shape_signs[]** and **gain_index[]**. The shape and gain indices are generated every subframe. The vector **shape_delay[]** contains the adaptive codebook lag for each subframe, while the vectors **shape_positions[]** and **shape_signs[]** contain the pulse positions and signs respectively. The adaptive codebook gain and multi-pulse gain for each subframe are vector quantized and stored in the vector **gain_index[]**.

3.B.9.2.2 Definitions

Input

PP_inputSignal[]: This array contains the preprocessed input signal and is of dimension sfrm_size.

int_Qlpc_coefficients[]: This array contains the interpolated and quantized LPC coefficients.

Wnum_coeff[]: This array contains the weighting filter coefficients of the numerator.

Wden_coeff[]: This array contains the weighting filter coefficients of the denominator.

Output

signal_mode: This field contains the voiced/unvoiced flag .

rms_index: This field defines the index for the signal power.

shape_delay[]: This array is of dimension *nrof_subframes*. It contains the codebook lag for the adaptive and fixed codebooks.

shape_positions[]: This array is of dimension *nrof_subframes*. It contains the pulse positions.

shape_signs[]: This array is of dimension *nrof_subframes*. It contains the pulse signs.

gain_index[]: This array is of dimension *nrof_subframes*. It contains the vector quantized gains for the adaptive codebook and the multi-pulse.

Configuration

frame_size: This field indicates the number of samples in one frame.

sbfrm_size: This field indicates the number of samples in one subframe.

nrof_subframes: This field indicates the number of subframes in one frame.

lpc_order: This field indicates the order of LPC.

3.B.9.2.3 Encoding process

For the Mode II coder, the excitation signal is extracted and encoded on an analysis-by-synthesis basis with three steps, the adaptive codebook search for a periodic component, the fixed codebook search for a non-periodic component and the quantization of the gains for each component. The target signal is obtained by subtracting the zero-input response of the synthesis and perceptual weighting filters from the weighted input speech signal.

Frame Energy Quantization

The root mean square (rms) value of subframe input samples is calculated at the last subframe. The rms value is scalar-quantized in the μ-law scale. The rms values of other subframes are obtained by linear-interpolating the quantized rms values at the last subframe of the current and the previous frames. The quantized rms values are used for gain normalization.

Open-loop Pitch Estimation and Mode Decision

The open-loop pitch estimation and mode decision procedures are jointly performed every analysis interval of 10 ms using the perceptually weighted signal. The pitch estimate is selected to minimize the square error between the current and previous blocks of the weighted input samples. A pitch prediction gain is calculated for each analysis interval. Every frame is classified into one of the four modes based on the average pitch prediction gain. Modes 0 and 1 correspond to unvoiced and transition frames, respectively. Modes 2 and 3 correspond to voiced frames, and the latter has higher periodicity than the former. If the subframe length is 5 ms, the same estimated pitch is used in two subframes for a closed-loop pitch search.

Encoding of Excitation Signal

The excitation signal is represented by a linear combination of the adaptive codevector and the fixed codevector scaled by their respective gains. Each component of the excitation signal is successively chosen by an analysis-by-synthesis search procedure so that the perceptually weighted error between the input signal and the reconstructed signal is minimized. The adaptive codevector parameters are a closed-loop delay and a gain. The closed-loop delay is selected in the range around the estimated open-loop delay. The adaptive codevector is generated from a block of the past excitation signal samples with the selected closed-loop delay. The fixed codevector contains several non-zero pulses. The pulse positions are restricted in an algebraic structure. The restriction table of the pulse positions are set up from the parameters. In order to improve the performance, after determining plural sets of pulse position candidates, a combination search between the pulse position candidates and the pulse amplitude index is carried out. The gains for both the adaptive and the multi-pulse codevectors are normalized and vector-

quantized. The normalization factor is calculated from the quantized LP coefficients, the quantized subframe rms and an excitation signal rms. The gain codebook is changed in accordance with the mode.

Adaptive Codebook Search

For each subframe the optimal delay is determined through closed-loop analysis so that the mean-square error between the target signal $x(n)$ and the weighted synthesized signal $y_t(n)$ of the adaptive codevector is minimized.

$$E_t = \sum_{n=0}^{N-1} (x(n) - g_t y_t(n))^2, \quad t_{op} - 8 < t < t_{op} + 8$$

where t_{op} is the open-loop delay determined in the open-loop pitch analysis. g_t is the optimal gain as follows:

$$g_t = \frac{\sum_{n=0}^{N-1} x(n) y_t(n)}{\sum_{n=0}^{N-1} y_t^2(n)}$$

The optimal pitch delay is encoded with 8 bits based on the relationship between the shape_indices[0] and the delay (see Decoding process).

Fixed Codebook Search

The fixed codebook vector is represented by the pulse position and pulse amplitudes. The pulse positions and amplitudes are searched to minimize the mean-square error between the target signal and the weighted synthesized signal.

$$E_k = \sum_{n=0}^{N-1} (x(n) - g_t y_t(n) - g_k z_k(n))^2$$

where g_t is the optimal gain as follows:

$$g_k = \frac{\sum_{n=0}^{N-1} (x(n) - g_t y_t(n)) z_k(n)}{\sum_{n=0}^{N-1} z_k^2(n)}$$

where k indicates the possible combination of the shape_indices[1] and the shape_indices[2] (see subclause 3.4.2). The fixed codebook vector is constructed from the pulse positions and the pulse amplitudes (see subclause 3.4.2). The weighted signal $z_k(n)$ is computed by filtering the fixed codebook vector through the LP synthesis filter $1/A(z)$ and the perceptual weighting filter $W(z)$.

Gain Quantization

Gains for the adaptive codevector and the fixed codevector are normalized by the prediction residual energy *rs* and scalar-quantized. The residual energy *rs* is calculated based on frame energy and reflection coefficients. The frame energy is calculated every subframe as a root mean square (RMS) value and quantized in the μ-law domain. The reflection coefficients *k*(*i*) are converted from the interpolated LPCs *int_Qlpc_coefficients*[]. Consequently, the prediction residual energy (*rs*) of a subframe is

$$rs = sbfrm\_len \cdot q\_amp^2 \cdot \prod_{i=1}^{N_p} \left(1 - k^2(i)\right),$$

where *Np* is the LP analysis order, *q_amp* is the quantized RMS amplitude and *sbfrm_len* is the subframe length.

The gain quantization is carried out by finding the pair of gain indices that minimize the error between the target and the weighted synthesized signal. The error err_g due to gain quantization is given by:

$$err_g = \sum_{i=0}^{sbfrm\_len-1} \left(\sqrt{\frac{rs}{pow_{AC}}} \cdot AdaptGainCB[m] \cdot ac\_syn(i) + \sqrt{\frac{rs}{pow_{SC}}} \cdot FixedGainCB[n] \cdot sc\_syn(i) - t\arg(i) \right)^2$$

$$pow\_ac = \sum_{i=0}^{sbfrm\_len-1} ac\_ex^2(i)$$

$$pow\_sc = \sum_{i=0}^{sbfrm\_len-1} sc\_ex^2(i)$$

where *m* and *n* are indices, *ac_ex*(*i*) is the selected adaptive codevector, *ac_syn*(*i*) is the signal which is synthesized from *ac_ex*(*i*) and perceptually weighted, *sc_ex*(*i*) is the selected stochastic codevector and *sc_syn*(*i*) is the signal which is synthesized from *sc_ex*(*i*)and perceptually weighted.

The index pair of *n* and *m* that minimize the error err_g is selected and is stored in *gain_indices*[]. Then the quantized adaptive gain *qacg*, the quantized stochastic *qscg* and the excitation signal *decoded_excitation*(*i*) are calculated as follows.

$$decoded\_excitation(i) = qacg \cdot ac\_ex(i) + qscg \cdot sc\_ex(i) \qquad (0 \le i \le sbfrm\_len - 1)$$

$$qacg = \sqrt{\frac{rs}{pow\_ac}} \cdot AdaptGainCB(m)$$

$$qscg = \sqrt{\frac{rs}{pow\_sc}} \cdot FixedGainCB(n)$$

3.B.9.3 Bitrate scalable multi-pulse excitation

3.B.9.3.1 Tool description

The output of the enhancement excitation analysis are **shape_enh_positions[]**, **shape_enh_signs[]** and **gain_enh_index[]**. The shape and gain indices are generated every subframe. The vectors **shape_enh_positions[]** and **shape_enh_signs[]** contain the enhancement pulse positions and signs respectively. The enhancement multi-pulse gain for each subframe is scalar quantized and stored in the vector **gain_index[]**.

3.B.9.3.2 Definitions

Input

PP_inputSignal[]: This array contains the preprocessed input signal and is of dimension sfrm_size.

int_Qlpc_coefficients[]: This array contains the interpolated and quantized LPC coefficients.

Wnum_coeff[]: This array contains the weighting filter coefficients of the numerator.

Wden_coeff[]: This array contains the weighting filter coefficients of the denominator.

Output

shape_enh_positions[]: This array is of dimension *nrof_subframes*. It contains the pulse positions.

shape_enh_signs[]: This array is of dimension *nrof_subframes*. It contains the pulse signs.

gain_enh_index[]: This array is of dimension *nrof_subframes*. It contains the vector quantized gains for the adaptive codebook and the multi-pulse.

Configuration

n_lag_candidates: This field indicates the number of lag candidates.

frame_size: This field indicates the number of samples in one frame.

sbfrm_size: This field indicates the number of samples in one subframe.

nrof_subframes: This field indicates the number of subframes in one frame.

lpc_order: This field indicates the order of LPC.

3.B.9.3.3 Encoding process

The fixed codebook vector is represented by the pulse position and pulse amplitudes. The pulse positions and amplitudes are searched to minimize the mean-square error between the target signal and the weighted synthesized signal.

$$E_k = \sum_{n=0}^{N-1} (x(n) - g_t y_t(n) - g_k z_k(n))^2$$

where g_t is the optimal gain as follows:

$$g_k = \frac{\sum_{n=0}^{N-1} (x(n) - g_t y_t(n)) z_k(n)}{\sum_{n=0}^{N-1} z_k^2(n)}$$

where k indicates the possible combination of the shape_enh_positions and the shape_enh_signs (see subclause 3.4.2). The fixed codebook vector is constructed from the pulse positions and the pulse amplitudes (see subclause 3.4.2). The weighted signal $z_k(n)$ is computed by filtering the fixed codebook vector through the LP synthesis filter $1/A(z)$ and the perceptual weighting filter $W(z)$.

3.B.9.4 Multi-pulse excitation for the bandwidth extension tool

3.B.9.4.1 Tool description

The output of the excitation analysis in the bandwidth extension tool are **shape_bws_delay[]**, **shape_bws_positions[]**, **shape_bws_signs[]** and **gain_bws_index[]**. The shape and gain indices are generated every subframe. The vector **shape_bws_delay[]** contains the adaptive codebook lag for each subframe, while the vectors **shape_bws_positions[]** and **shape_bws_signs[]** contain the pulse positions and signs respectively. The adaptive codebook gain and multi-pulse gain for each subframe are vector quantized and stored in the vector **gain_bws_index[]**.

3.B.9.4.2 Definitions

Input

PP_inputSignal[]: This array contains the preprocessed input signal and is of dimension sfrm_size.

int_Qlpc_coefficients[]: This array contains the interpolated and quantized LPC coefficients.

Wnum_coeff[]: This array contains the weighting filter coefficients of the numerator.

Wden_coeff[]: This array contains the weighting filter coefficients of the denominator.

Output

shape_bws_delay[]: This array is of dimension *nrof_subframes_bws*. It contains the codebook lag for the adaptive and fixed codebooks.

shape_bws_positions[]: This array is of dimension *nrof_subframes_bws*. It contains the pulse positions.

shape_bws_signs[]: This array is of dimension *nrof_subframes_bws*. It contains the pulse signs.

gain_bws_index[]: This array is of dimension *nrof_subframes_bws*. It contains the vector quantized gains for the adaptive codebook and the multi-pulse.

Configuration

n_lag_candidates: This field indicates the number of lag candidates.

frame_size: This field indicates the number of samples in one frame.

sbfrm_size: This field indicates the number of samples in one subframe in the bandwidth extension tool.

nrof_subframes_bws: This field indicates the number of subframes in the bandwidth extension tool.

lpc_order: This field indicates the order of LPC.

3.B.9.4.3 Encoding process

For the bandwidth extension tool, the excitation signal is extracted and encoded on an analysis-by-synthesis basis with three steps, the adaptive codebook search for a periodic component, the fixed codebook search for a non-periodic component and the quantization of the gains for each component. The target signal is obtained by subtracting the zero-input response of the synthesis and perceptual weighting filters from the weighted input speech signal.

Frame Energy Quantization

The same root mean square (rms) value as the 8 kHz sampling rate CELP is used.

Open-loop Pitch Estimation and Mode Decision

The open-loop pitch estimation is done by converting the 8 kHz sampling rate pitch delay (see Decoding process). The same mode as the 8 kHz sampling rate CELP is used.

Encoding of Excitation Signal

The excitation signal is represented by a linear combination of the adaptive codevector and the two fixed codevectors scaled by their respective gains. The pitch delay is decided in the range around the open-loop estimated pitch delay. One of the two fixed codevectors is obtained by sampling-rate expansion of the fixed codevector used in the 8 kHz sampling rate coder. The other fixed codevector is determined by an analysis-by-synthesis search procedure.

Adaptive Codebook Search

For each subframe the optimal delay is determined through closed-loop analysis so that the mean-square error between the target signal $x(n)$ and the weighted synthesized signal $y_t(n)$ of the adaptive codevector is minimized.

$$E_t = \sum_{n=0}^{N-1} (x(n) - g_t y_t(n))^2, \quad t_{op} - 8 < t < t_{op} + 8$$

where t_{op} is the open-loop delay determined in the open-loop pitch analysis. g_t is the optimal gain as follows:

$$g_t = \frac{\sum_{n=0}^{N-1} x(n) y_t(n)}{\sum_{n=0}^{N-1} y_t^2(n)}$$

The difference between the optimal pitch delay and the open-loop pitch delay is encoded with 3 bits based on the relationship between the shape_bws_delay and the differential delay (see Decoding process).

Fixed Codebook 1 Search

The fixed codevector 1 is obtained by sampling-rate expansion of the fixed codevector used in the 8 kHz sampling rate coder (see subclause 3.5.7.4.3.4).

Fixed Codebook 2 Search

The fixed codebook vector is represented by the pulse position and pulse amplitudes. The pulse positions and amplitudes are searched to minimize the mean-square error between the target signal and the weighted synthesized signal.

$$E_k = \sum_{n=0}^{N-1} (x(n) - g_t y_t(n) - g_k z_k(n))^2$$

where g_t is the optimal gain as follows:

$$g_k = \frac{\sum_{n=0}^{N-1}(x(n) - g_t y_t(n))z_k(n)}{\sum_{n=0}^{N-1} z_k^2(n)}$$

where k indicates the possible combination of the shape_bws_positions and the shape_bws_signs (see subclause 3.4.2). The fixed codebook vector is constructed from the pulse positions and the pulse amplitudes (see subclause 3.4.2). The weighted signal $z_k(n)$ is computed by filtering the fixed codebook vector through the LP synthesis filter $1/A(z)$ and the perceptual weighting filter $W(z)$.

Gain Quantization

Gains for the adaptive codevector and the two fixed codevectors are normalized by the prediction residual energy *rs* and quantized. The residual energy *rs* is calculated based on frame energy and reflection coefficients. The frame energy is calculated every subframe as a root mean square (RMS) value and quantized in the μ-law domain. The reflection coefficients *k*(*i*) are converted from the interpolated LPCs *int_Qlpc_coefficients*[]. Consequently, the prediction residual energy (*rs*) of a subframe is

$$rs = sbfrm\_len \cdot q\_amp^2 \cdot \prod_{i=1}^{N_p}\left(1 - k^2(i)\right),$$

where *Np* is the LP analysis order, *q_amp* is the quantized RMS amplitude and *sbfrm_len* is the subframe length.

The gains for the adaptive codevector and the fixed codevector 2 are vector quantized. The gain for the fixed codevector 1 is scalar quantized. These quantization operations are achieved by minimizing the perceptually weighted distortion (closed-loop).

3.B.10 CELP bitstream multiplexer

3.B.10.1 Tool description

The tool CELP bitstream multiplexer multiplexes a frame into the bitstream.

3.B.10.2 Definitions

All the bitstream elements and the helping variables have been defined in subclause 3.5.3 and subclause 3.5.4.

3.B.10.3 Encoding process

The parameters are encoded into a bitstream according to the syntax described in subclause 3.3.

3.B.11 CELP silence compression tool

3.B.11.1 VAD module

The VAD module makes a decision whether a frame is a non-active-voice frame or an active-voice frame based on how much the characteristics of the input signal change. The characteristics are represented by four parameters: the full-band and the low-band energies, the LSPs, and the zero-crossing rate of the input signal frame. A temporal decision is made in every 80 samples and the final frame decision is made based on the temporal decisions with a hangover constraint. For the wideband mode, the characteristics parameters are calculated from the input speech down-sampled from 16 kHz to 8 kHz.

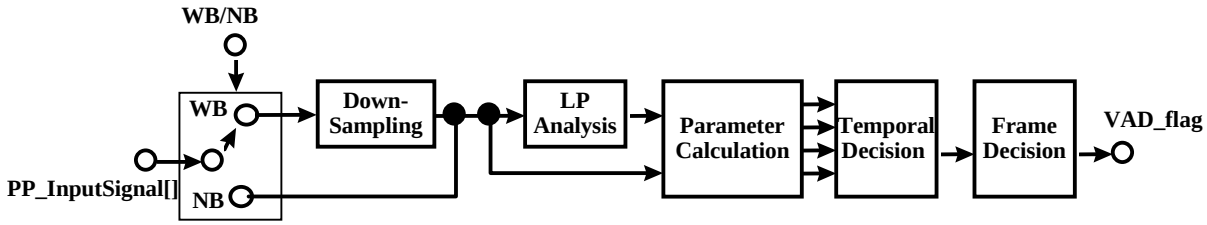


Figure 3.B.1 — VAD module

3.B.11.1.1 Definitions

Input

PP_InputSignal[] This array contains the pre-processed speech signal. The dimension is *frame_size*.

Output

VAD_flag This field contains the VAD flag (see Table 3.103).

The following are help elements used in the VAD module:

lpc_order: the order of LP

sbfrm_size: the number of samples in a subframe

frame_size: the number of samples in a frame

n_subframe: the number of subframes in a frame

3.B.11.1.2 Down-sampling for the wideband

A signal *s_vad*[] used in the VAD module is generated by preprocessing the input signal in the same manner as that described in subclause 3.B.4. When the sampling rate is *16 kHz*, the input signal is down-sampled to *8 kHz* after the preprocessing.

3.B.11.1.3 Parameter calculation

The LSPs of the input signal, *lsp*[] are calculated from the LPCs *lpc_coefficients*[], which is given by the MPEG-4 CELP weighting module described in subclause 3.B.8. A full-band energy *P*, a low-band energy P_l between 0 and 1 kHz, and a zero crossing rate *Z* are calculated as follows:

$$P = 10\ log_{10} R[0]$$

$$P_l = 10\ log_{10} \mathbf{h}^T \mathbf{R} \mathbf{h}$$

$$Z = \frac{1}{80} \sum_{i=0}^{80} \left| sign\left[s\_vad[i]\right] - sign\left[s\_vad[i-1]\right] \right| ,$$

where ***h*** is an impulse response vector of the FIR filter with a cutoff frequency of *1 kHz. R*[*0*] is the first autocorrelation coefficient, and ***R*** is a Toeplitz autocorrelation matrix with autocorrelation coefficients in each diagonal. These parameters are calculated every *10 msec*. Their averages are updated as follows:

$$\overline{P} = a\,\overline{P} + (1-a)\,P$$
$$\overline{P_l} = b\,\overline{P_l} + (1-b)\,P_l$$
$$\overline{Z} = c\,\overline{Z} + (1-c)\,Z$$
$$\overline{Lsp}[i] = d\,\overline{Lsp}[i] + (1-d)\,Lsp[i]\text{, } i=1,\ldots lpc\_order$$

where *a = 0.995, b = 0.995, c = 0.998* and *d = 0.75*. The following differential parameters are evaluated to make a temporal decision whether the input signal is characterized as an active-voice or a non-active-voice every *80 samples*:

$$\Delta P = \overline{P} - P$$
$$\Delta P_l = \overline{P_l} - P_l$$
$$\Delta Z = \overline{Z} - Z$$
$$\Delta Lsp = \sum_{i=1}^{lpc\_order} \left[\overline{Lsp}[i] - Lsp[i]\right]^2\text{, } i=1,\ldots lpc\_order.$$

3.B.11.1.4 Temporal voice activity decision

If any of the following inequalities is satisfied, the temporal voice activity flag, *vad_flag_sub*[*i*] *for* $i=0,\ldots,\frac{frm\_size[samples]}{80[samples]}$, is set to 1.

if(
ΔLsp *> 0.0009 or*
ΔLsp *> 0.00175 * ΔZ + 0.00085 or*
ΔLsp *> -0.00455 * ΔZ + 0.00116 or*
ΔP *< -0.47 or*
ΔP *< -2.5 * ΔZ - 0.5 or*
ΔP *< 2.0 * ΔZ - 0.6 or*
ΔP *< 2.5 * ΔZ - 0.7 or*
ΔP *< -2.91 * ΔZ - 0.482 or*
ΔP *< 880.0 * ΔLsp - 1.22 or*
ΔP_l *< 1400.0 * ΔLsp - 1.55 or*
ΔP_l *> 0.929 * ΔP + 0.114 or*
ΔP_l *< -1.5 * ΔP - 0.9 or*
ΔP_l *< 0.714 * ΔP - 0.214*
)
{
vad_flag_sub[i] = 1;
} else {
vad_flag_sub[i] = 0;
}

3.B.11.1.5 Frame voice activity decision

A frame voice activity flag, *vad_flag* is determined based on the temporal flag *vad_flag_sub*[], which is made in the corresponding frame as follows:

vad_flag = *vad_flag_sub[0]* ∪ *vad_flag_sub[1]* ∪ … ∪ *vad_flag_sub[L-1],*

where "∪" stands for a logical OR.

3.B.11.1.6 Hangover

A hangover is applied to the final VAD decision *VAD_flag* after the switch from an active-voice frame to a non-active-voice frame:

$$VAD\_flag = \begin{cases} 1, & first\ 80\ msec\ after\ the\ active-voice\ period\ (vad\_flag = 1) \\ vad\_flag, & otherwise \end{cases}$$

When the active-voice period (*vad_flag=1*) is shorter than *80 msec*, *VAD_flag* is always equal to *vad_flag.*

3.B.11.2 DTX module

Figure 3.B.2 shows the structure of the DTX module. The DTX module detects frames in which the input characteristics change during the non-active-voice frames. In the first frame during each non-active-voice period and in the frame where the change is detected, the DTX module extracts the parameters; the frame energy and the LSPs of the input speech, and encodes these parameters. There are three DTX modes; DTX_flag = 0. 1 and 2 depending on what information is transmitted. When a change in the LSPs is detected (DTX_flag = 1), the encoded LSP and RMS parameters and the TX_flag are transmitted as HR-SID information. When a change of the RMS (the frame energy) is detected (DTX_flag = 2), only the encoded RMS parameter and the TX_flag are transmitted as LR-SID information. Otherwise, only the TX_flag is transmitted.

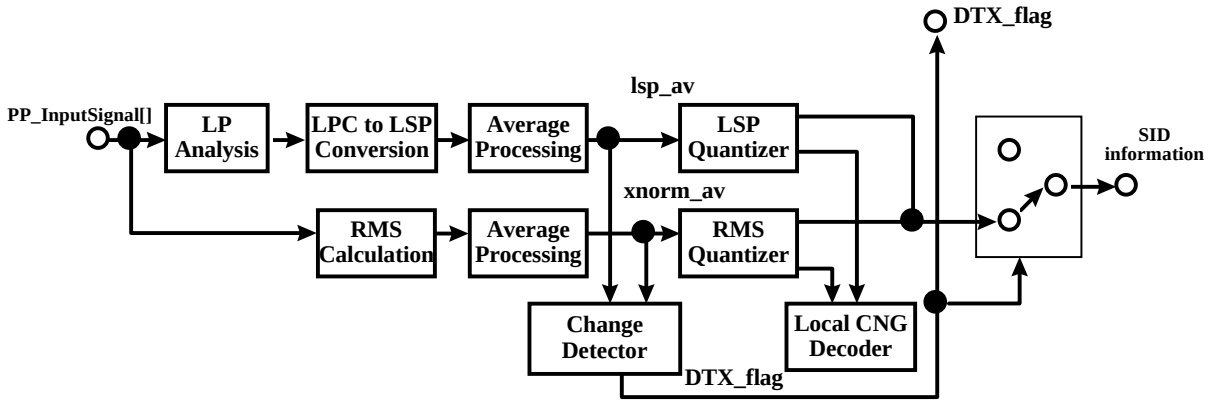


Figure 3.B.2 – DTX module

3.B.11.2.1 Definitions

Input

PP_InputSignal[] This array contains the pre-processed speech signal. The dimension is *frame_size*.

TX_flag This field contains the transmission mode (see Table 3.103).

Output

VAD_flag This field contains the VAD flag (see Table 3.103).

The following are help elements used in the DTX module:

lpc_order: the order of LP

frame_size: the number of samples in a frame

n_subframe: the number of subframes in a frame

3.B.11.2.2 LP analysis

The same LP analysis and LPC-to-LSP conversion as those described in subclause 3.B.5 is used. This gives unquantized LSPs *lsp*[][].

3.B.11.2.3 Averaging of the LSP

An average LSP *lsp_av*[] is calculated from the unquantized LSPs *lsp*[][] as follows:

$$lsp\_av[i] = (1/L)\sum_{j=0}^{L-1} lsp[j][i], \qquad i=0,...,lpc\_order-1.$$

where *lsp*[j][] are unquantized LSPs in the *j-th* most recent frame, which is calculated from the unquantized LPCs *lpc_coefficients*[] described in subclause 3.B.6. *L* is the number of frames per *80 msec*.

3.B.11.2.4 RMS calculation

A RMS of the input signal is calculated in an identical manner to that described in subclause 3.B.9. This calculation gives unquantized RMS *xnorm*[] in each subframe.

3.B.11.2.5 Averaging of the RMS

An average RMS of the input signal, *xnorm_av*[] is computed from the unquantized RMS *xnorm*[] as follows:

$$xnorm\_av = (1/M)\sum_{j=0}^{M-1} xnorm[j]$$

where *M* is the number of subframes per *80 msec*. Also, *xnorm_av_end* is *xnorm_av*[*n_subframe-1*], where *n_subframe* is the number of subframes in a frame.

3.B.11.2.6 Detection of the characteristics change

Any change in the characteristics is detected based on the variations in the frame energy and the spectrum computed from the input signal as follows:

DTX_flag = 0

if($\left|20\log_{10} xnorm\_sid - 20\log_{10} xnorm\_end\right| > D_{xnorm}\ dB$ *)DTX_flag=2*

if($\sum_{i=1}^{lpc\_order}\left[lsp\_sid[i] - lsp\_av[i]\right]^2 > D_{lsp}$ *) DTX_flag=1,*

where LSPs are normalized to a range from 0 to 1. A threshold D_{xnorm} is switched according to *xnorm_sid* as shown in Table 3.B.14. A threshold D_{lsp} is changed according to the sampling rate; *0.002* for *8 kHz* and *0.0015* for *16 kHz*. *lsp_av_sid* and *xnorm_sid* are *lsp_av* and *xnorm_av_end* of the last SID frame, respectively. There is a minimum period where the detection is not made. The length of the period is normally *20 msec*, but it is *0 msec* during the first *40 msec* of the non-active-voice period.

Table 3.B.14 — Relationship between D_{xnorm} and *xnorm_sid*

$20log_{10} xnorm\_sid$ [dB]	D_{xnorm} [dB]
< 1.0:	6.0
1.0 ~ 5.0	4.0
5.0 ~ 9.0	3.0
9.0 ~ 12.0	2.5
12.0 <	2.0

3.B.11.2.7 Parameter encoding

The average LSP *lsp_av*[] is encoded using the same process as that described in subclause 3.B.6 with the exceptions described in subclause 3.5.9.3. The encoding of the average RMS *xnorm_av*[] is identical to that described in subclause 3.B.9, with the exceptions that the μ-law parameters are independent of the signal mode and are set as *rms_max = 7932* and *mu_law = 1024*.

3.B.11.2.8 Local CNG decoder

Local CNG decoding is performed to update buffers for the LP synthesis filter. The processing is identical to the decoding process in the decoder.

Annex 3.C
(normative)

Tables

3.C.1 LSP VQ tables and gain VQ tables for 8 kHz sampling rate

Table 3.C.1 — LSP table for the first stage (lsp_tbl[0][Index][])

scale_factor = 2^15

Index	Codeword				
	0	1	2	3	4
0	2465	3900	5664	7874	9529
1	2430	4275	6343	8750	11411
2	1911	2913	4284	6757	12988
3	2978	5317	8371	11095	14258
4	1878	2844	4285	6524	8539
5	2382	3849	5636	8795	13807
6	1171	1944	4453	9571	12822
7	3212	6003	9409	12522	15568
8	1755	2837	4879	8370	11087
9	2987	5164	7422	9876	12412
10	1895	2870	4047	7313	15485
11	1749	3596	7928	12060	15520
12	1545	2253	3430	5844	11194
13	1816	3463	6818	10315	13176
14	1919	2912	4709	10681	15270
15	3405	6578	11374	14885	17519

Table 3.C.2 — LSP table for the first stage (lsp_tbl[1][Index][])

scale_factor = 2^15

Index	Codeword				
	0	1	2	3	4
0	16725	19702	22552	25746	27870
1	14891	18400	21563	24768	29116
2	18815	21093	23260	25905	28001
3	14648	18159	21499	24676	27025
4	16274	19321	22413	26092	29415
5	11847	19451	22070	25544	28584
6	20171	22959	25232	27632	29514
7	12579	16636	20917	23824	26623
8	16685	18912	21352	24430	27213
9	14427	18039	20139	26405	28635
10	17223	20497	23557	26856	29546
11	14627	17482	19981	22769	26192
12	14411	19945	22550	26869	29219
13	11882	16454	21931	25698	28836
14	18194	21846	24892	27871	29977
15	13161	16571	19143	23542	28717

Table 3.C.3 — LSP table for the second stage of VQ without interframe prediction (d_tbl[0][Index][])

scale_factor = 2^18

Index	Codeword				
	0	1	2	3	4
0	-1126	-109	2708	-1644	4815
1	-2453	2636	8784	5571	-1482
2	-1303	5382	2696	7113	-4855
3	2833	-70	915	1226	-223
4	3977	1246	-3288	-719	6271
5	3881	5460	5880	3784	-2910
6	-465	443	-2027	4891	2781
7	7186	8149	-6311	785	766
8	3509	4478	5088	354	3239
9	-2612	-1774	8384	2234	3546
10	455	-565	17	10318	-2947
11	3618	6966	7303	-7657	4683
12	-3251	-3354	-2990	6106	10100
13	3272	4496	5938	-3011	-3413
14	-1571	-2953	-4055	10707	3887
15	6610	11258	-11073	-1875	9146
16	661	5993	-567	-2306	8104
17	-2670	8858	6054	-96	1956
18	-4109	-3480	3662	11658	-1416
19	-200	425	3682	4465	-3850
10	6475	-1500	5350	6261	570
21	3196	3229	8497	2406	-8489
22	751	3263	3153	6347	2053
23	4730	7170	-728	3346	-5906
24	1520	1650	1984	5703	7990
25	-4481	-4064	15246	6561	-4680
26	1745	3676	7627	12515	177
27	2901	2443	2531	-5361	2756
28	988	-4549	-1624	404	9058
29	6914	11203	8517	683	-1274
30	-3178	-6034	10596	20414	12845
31	6578	8953	-2591	-7687	6476
32	4211	104	3424	-301	8485
33	-2034	1585	11201	-5153	1532
34	1639	2492	5533	10513	-8746
35	2358	5574	-808	-495	-316
36	3080	3666	-5363	6444	6223
37	9041	1554	2776	1534	-8749
38	-1962	-3634	1107	5018	4525
39	6003	9151	552	2144	5896
40	3118	4920	10734	3848	3150
41	-4584	-3459	4470	988	10814
42	4337	2406	-5712	8307	-3022
43	-675	3459	7994	-2264	9644
44	-1961	960	-1511	1664	8638
45	5874	10566	6805	-12232	-4384
46	1303	1031	231	12526	3738
47	11179	1372	-10700	1154	9015
48	573	758	906	-7236	10705
49	-5865	14935	10133	563	-8644
50	-2600	-2868	6607	8258	4025
51	244	-983	7562	-440	-2133
52	6886	-5508	-2676	5119	4081
53	-1360	4510	12050	-1224	-5202
54	-6719	3411	2672	5165	3416

55	6765	10685	-3144	-5258	-5219
56	891	4010	6076	11065	12632
57	-6906	-6860	13524	3954	4216
58	6686	6774	894	9019	380
59	11611	2335	203	-1753	1440
60	1729	725	-5941	624	13626
61	9827	13514	8008	-2229	-11638
62	-2343	-7659	-7405	15476	14625
63	16104	22055	-1580	110	3939

Table 3.C.4 — LSP table for the second stage of VQ without interframe prediction (d_tbl[1][Index][])

scale_factor = 2^18

Index	Codeword				
	0	1	2	3	4
0	-784	4977	10919	2178	-7243
1	-1182	-3519	5460	3383	6735
2	2546	6242	2843	2291	6144
3	6510	-2312	3823	-3920	5884
4	-166	1153	4178	2694	-24
5	-6243	6349	-1005	7150	2470
6	2245	6605	2211	867	-3388
7	326	341	-2498	2721	5056
8	7981	2249	3442	4998	2602
9	-6443	-87	11399	890	2895
10	3935	1676	-5725	10571	2817
11	1083	-1790	-950	1243	15407
12	5676	7745	-1065	-6090	5950
13	-383	7422	7295	8602	2479
14	6200	768	8053	-2876	-2433
15	2730	-6624	2061	11171	923
16	-496	1617	4072	10057	-7194
17	4849	-5669	11789	3548	-816
18	-911	8873	-5233	1736	8291
19	-3367	4160	5035	-4114	7573
20	5630	810	-1134	573	-292
21	-6603	-703	5740	10320	2070
22	2973	12420	4445	-4076	-5042
23	10283	622	-7195	1350	7339
24	10608	-1538	1579	5882	-6505
25	-8052	10656	7008	-292	1287
26	4279	8612	-5498	5182	-3520
27	-5223	-2292	523	9511	13373
28	11418	5020	422	-4211	-911
29	8496	7600	10246	12203	12798
30	2299	5319	13330	-8096	1792
31	7693	-7706	-249	4207	6541

Table 3.C.5 — LSP table for the second stage of VQ with interframe prediction (pd_tbl[0][Index][])

scale_factor = 2^19

Index	Codeword				
	0	1	2	3	4
0	2532	3998	7136	9485	5083
1	-2097	5173	3711	4349	-2269
2	3145	936	-3045	4502	3310
3	-2425	-3497	2046	7254	4988

4	-1978	-2428	-5680	11862	7647
5	4003	306	4245	-2666	1903
6	1961	-2177	1126	120	9677
7	-1550	6306	8296	-4272	1462
8	1651	1603	5298	10637	-10274
9	3951	10014	1254	4008	-10934
10	6296	9279	-7936	1487	3241
11	-2658	-794	12908	609	-3544
12	631	1808	4942	10243	14943
13	6384	8886	7088	-543	7770
14	-707	482	6315	-6654	11595
15	-11649	12236	13527	9612	-2440
16	4684	5285	1631	18867	6223
17	3528	4132	7496	-15	-7553
18	2181	360	-3133	12049	-1862
19	-2449	-1085	9723	2395	6722
20	-560	-3142	-6080	28195	10042
21	4649	5443	-54	-8917	3778
22	-2700	-4856	-5676	10230	23087
23	4181	6107	16421	-6980	887
24	-4017	-2248	10625	12374	-2242
25	12560	13793	6109	843	-11284
26	16905	2231	-1388	-3055	2074
27	-2914	-1769	27848	7691	-3565
28	2504	5855	4073	3666	25794
29	16701	15278	9923	8078	1800
30	4846	4518	-3418	-5273	13037
31	4344	7195	13937	5995	-1118
32	6069	9436	4428	9661	-1595
33	2587	-1877	4404	5849	-2742
34	-2951	10357	-5009	10431	-1785
35	9470	-8852	10152	4690	-1396
36	5616	1774	-12799	12062	7965
37	-1384	6827	514	732	6505
38	-2066	990	-5485	2642	14029
39	6054	9511	5459	-6155	-3512
40	3894	5517	7460	19731	-12932
41	6945	6225	-1180	807	-3202
42	8980	9279	-16470	-2140	14198
43	-6027	-2368	20039	-3642	6502
44	6729	5214	-821	8497	9492
45	3454	8769	17467	847	14653
46	4004	7837	6595	-14235	9321
47	-3091	23678	7283	-149	8
48	4511	6737	14600	21923	4316
49	2426	8975	16750	1587	-14478
50	12018	6049	-5398	11537	-6297
51	-12145	8219	5436	-1020	10428
52	-3522	-6973	2794	16587	9151
53	9935	16017	-3390	-8287	1157
54	-5035	-5864	3814	2300	16845
55	7871	13250	13574	-14814	-5398
56	-1223	-1589	2711	21386	-3234
57	24577	29116	4713	-6772	-11585
58	10902	-8650	-4718	3868	10261
59	-6979	-10767	20809	14282	10494
60	3887	7227	11996	22138	24134
61	10278	16482	-939	5802	13611
62	924	-278	-2632	-11433	23190

63	6606	19360	25601	8376	4412

Table 3.C.6 — LSP table for the second stage of VQ with interframe prediction (pd_tbl[1][Index][])

scale_factor = 2^19

Index	Codeword				
	0	1	2	3	4
0	6216	-10079	2425	14328	263
1	444	660	4677	5548	1890
2	-9634	3530	-1050	15523	4278
3	-7425	10943	-3973	378	14153
4	1564	4930	8272	23911	-7335
5	-3677	13414	8730	-1472	-6113
6	-8226	4572	12153	-2383	10206
7	5960	5150	-8113	11969	176
8	9559	1543	-1145	1563	-2662
9	19979	1828	9044	10797	90
10	-6465	-6562	1897	5629	15698
11	5548	6185	4786	-5789	4433
12	2335	-3422	15593	5129	-6218
13	-250	3805	20165	-10762	2234
14	4761	8374	7904	10727	14375
15	2897	14078	-5945	-3216	1636
16	20354	-10762	-5133	5894	2845
17	8021	-7160	10153	-3581	7498
18	1747	-3572	-8747	21621	14054
19	7631	1608	-8888	2778	12518
20	10893	13976	3339	6689	-11601
21	7958	24421	15064	-14485	795
22	-19013	20319	15065	12309	6725
23	-3356	14533	498	9132	-1070
24	22372	6674	-5593	-7807	2231
25	17496	3465	14129	-6401	-7937
26	10957	-11012	5450	8484	19068
27	8076	4615	132	-13695	17504
28	-7926	-6788	19762	13593	5532
29	3638	11875	23191	6498	416
30	10807	25123	27312	12296	21037
31	11303	22562	-3922	6445	9484

Table 3.C.7 — Gain VQ tables for narrowband CELP coder (subframe 10 ms)

scale_factor = 2^14

	Mode 0		Mode 1		Mode 2		Mode 3	
Index	Codeword		Codeword		Codeword		Codeword	
	0	1	0	1	0	1	0	1
0	96	281	368	936	1306	2931	3022	6368
1	845	8737	5856	5202	7476	4118	9583	2479
2	871	6180	986	9557	5780	4373	7846	2595
3	4506	9494	6509	10788	12130	3716	11850	2593
4	709	4694	967	7335	5905	12020	1480	10720
5	706	10947	8873	6200	9709	3898	12085	5902
6	3881	6269	1282	15500	8719	6102	9043	4222
7	1279	16019	10471	17218	14796	9719	14434	3000
8	716	3313	1125	5115	1100	9653	6162	3034
9	3065	9462	6809	8187	9908	6467	11563	4303

10	3005	8360	4195	9873	3435	8728	10165	4356
11	7511	9871	8713	9682	15981	5240	12613	3700
12	2311	4285	3743	7146	2705	17876	7575	5897
13	3107	12901	11690	6429	11973	5831	14900	5233
14	5864	8939	2797	22418	9299	8816	9774	8479
15	5868	19081	19571	9804	23501	8763	17281	6495
16	557	2243	772	3424	1928	6882	6070	5420
17	2011	9153	7457	5735	9521	5312	10837	4086
18	1825	7239	925	12834	7829	5600	9652	3576
19	5558	11154	7986	12868	13957	4168	12966	2753
20	1952	6072	2791	9451	9505	11808	4916	12059
21	594	12315	10061	7133	10872	5904	12947	6324
22	5426	6526	4846	15527	8007	8422	9093	5935
23	3158	22812	14455	15456	13886	14573	15696	4124
24	2101	2949	3656	3962	1060	13149	7980	4211
25	2963	11141	8574	8122	11217	6998	12292	4651
26	4469	7696	4343	12456	5831	7830	10306	5719
27	10380	14515	12104	10043	17088	8137	13752	4874
28	3604	4931	5513	8467	8486	19002	7582	9357
29	5220	13359	14517	8918	13544	7879	14949	8191
30	7338	7999	7715	21994	11773	8537	11595	9334
31	9446	26080	20331	24179	22611	14536	23132	8850
32	493	1250	1037	2067	3707	4104	4637	3690
33	760	9893	5958	6948	8662	4462	10800	2566
34	764	7523	1116	11096	6600	5919	8773	3183
35	4708	10373	7971	10992	12729	4786	11969	3522
36	1639	5099	2369	7838	7255	14492	3708	9061
37	1750	11533	10381	5297	10847	3952	12039	7252
38	3557	7381	2307	18621	8907	7296	9639	4995
39	2017	18755	14314	20305	16349	12224	14379	4154
40	1426	3783	2609	5638	2162	10979	7052	3907
41	3637	10224	7307	9336	10293	7768	11653	5121
42	3775	8693	4999	10883	4576	10169	10873	5023
43	8861	11649	10093	10929	18764	6195	13094	4351
44	2766	5362	4448	8137	3795	25643	8457	7271
45	4064	15446	14209	6152	13187	6012	15496	6400
46	6091	9979	3392	29021	10528	9754	10810	7304
47	9807	19395	26110	13673	28108	13457	19747	6402
48	1323	2190	2146	3452	4667	6562	6163	7827
49	2185	10203	7695	7195	10418	5010	11242	3436
50	2308	7885	2734	13459	7471	7123	10368	3358
51	6892	11954	9775	13841	14328	5757	13550	3585
52	2820	6600	2876	11267	11101	13711	7341	20130
53	1252	13617	11657	8071	11569	4879	13531	7617
54	5622	7800	7343	16406	7996	10345	9750	6737
55	2966	28962	19660	15528	17861	17087	17267	4304
56	3336	3610	4366	5732	3443	13095	8562	5111
57	4297	11656	10011	8757	12345	7093	12939	5312
58	4812	8669	6094	12674	6487	9379	11206	6105
59	15465	16210	12367	12693	19647	10249	14038	6109
60	4959	5156	5834	9531	13310	23800	9634	11944
61	7145	14652	16105	12086	15149	7295	17026	9771
62	10026	9255	12981	28541	12619	10600	13830	10845
63	19180	26086	27798	24140	25784	23526	22790	15754

Table 3.C.8 — Gain VQ tables for narrowband CELP coder (subframe 5 ms)

scale_factor = 2^14

	Mode 0		Mode 1		Mode 2		Mode 3	
Index	Codeword		Codeword		Codeword		Codeword	
	0	1	0	1	0	1	0	1
0	391	1069	961	2363	1259	3029	1668	2677
1	5628	8023	8086	10164	7429	3747	9023	1963
2	829	8075	1156	9804	1329	10359	7434	2339
3	4140	12001	12362	4422	12439	2804	13857	2348
4	2737	7627	4699	4094	5954	3003	2904	5219
5	6954	10082	7567	4483	10677	3109	12795	2244
6	605	11822	1718	15709	8353	8941	10352	4793
7	1641	23690	19114	6013	17744	4540	18047	3357
8	915	5938	1056	6890	1308	6536	5974	2422
9	6832	11666	11421	10629	10975	4609	10554	2002
10	2738	9942	3921	9479	3023	17711	7977	6447
11	5245	16314	14333	7447	14874	3557	16346	2783
12	4451	4950	5972	9629	7137	5548	6258	10555
13	9248	11696	10037	7031	12123	6395	13598	3833
14	1108	14505	2485	24553	13258	7967	12066	4764
15	9896	17912	19492	12937	25939	9249	23084	5066
16	1237	4207	2899	3481	3041	2522	4664	2572
17	5410	9776	7877	13160	8970	3281	10241	3399
18	580	10144	1026	12705	5216	13404	9338	3486
19	4300	13531	14297	4156	12858	3999	14940	2521
20	4145	7719	6914	6858	4521	5187	2490	9940
21	9812	9792	10276	4339	12021	4928	12790	3475
22	2725	11800	7859	16648	10923	12399	11570	3269
23	7439	23129	23881	7023	23428	5987	18230	5831
24	2542	5383	4367	6669	4047	10576	7302	4612
25	6683	13086	12151	16983	10998	7464	11722	2116
26	4040	9861	5580	12745	5804	25533	11106	6224
27	9643	14815	15398	10431	16073	5571	15755	4799
28	7251	6897	8435	7460	9627	5713	4223	20978
29	13087	12577	12206	7124	14679	5729	14597	4329
30	1583	17435	11352	23655	17201	9850	13704	6495
31	17862	21631	21928	18475	23768	15557	28103	5726
32	1086	2604	1268	4528	2766	4553	3367	2373
33	6469	8905	9508	11003	8530	4977	9815	2508
34	1789	9034	2644	11119	1703	13131	8312	2981
35	5401	12323	13316	5662	13683	3154	14272	3294
36	3326	8775	5811	5419	5951	4597	5456	6165
37	8041	10614	8917	5781	11722	3697	13293	2960
38	1419	12776	2697	19079	10012	9407	11288	4662
39	4159	28738	19686	9274	20011	5339	19885	4449
40	1901	6811	2751	7956	3462	7679	6681	3515
41	8160	12688	13169	12490	10860	5906	11013	2739
42	3468	10893	4638	11131	8584	17996	9664	7087
43	6930	18050	16591	8026	16056	3835	16828	4183
44	5196	6583	6736	11176	8007	6954	9638	13184
45	10702	12680	10952	8527	13506	6436	13697	4894
46	3165	15136	5373	28942	14178	10526	12509	5786
47	12581	20934	24291	12982	29840	8673	25353	8792
48	2697	3664	2949	5624	4445	3303	5300	3932
49	5964	10859	10292	13748	9922	4299	10959	3752
50	1801	10841	3357	13330	7808	11753	9382	4964
51	5642	14198	16184	5465	14082	4498	15360	3579
52	4643	8881	7335	8619	5797	6905	2593	14049

53	11948	8660	11005	5798	13093	5209	12874	4452
54	2959	13080	7186	20738	14391	15928	12077	3842
55	11554	27975	29648	11305	21401	8985	20987	8017
56	3619	6425	5338	8052	6216	9336	8459	4302
57	7612	14974	16002	19976	11916	9009	12211	2850
58	4763	10908	5258	15290	16067	25296	11966	8220
59	12680	16015	16309	14492	17912	6929	16506	6487
60	8164	8664	9310	8810	9607	7222	9287	25100
61	18715	14800	12917	9000	15344	7661	14840	5640
62	3702	19285	17523	27956	19372	13134	14961	9721
63	21995	27675	27604	24296	29648	18568	30284	10468

Table 3.C.9 — Filter coefficients for adaptive codebook interpolation (for 8 kHz sampling rate)

scale_factor = 2^15

Coefficients.	Value
0	32767
1	31236
2	26910
3	20534
4	13174
5	5989
6	0
7	-4097
8	-6045
9	-6016
10	-4528
11	-2286
12	0
13	1768
14	2700
15	2750
16	2100
17	1068
18	0
19	-822
20	-1246
21	-1253
22	-942
23	-469
24	0
25	346
26	509
27	498
28	363
29	176
30	0
31	-123
32	-181
33	-180
34	-138
35	-72
36	0

3.C.2 LSP VQ tables and gain VQ tables for the 16 kHz sampling rate

Table 3.C.10 — Table for the first stage, lower part of the wideband VQ (lsp_tbl[0][Index][])

scale_factor = 2^15

Index	Codeword				
	0	1	2	3	4
0	1143	1850	2663	3727	5954
1	1190	1963	2858	4886	6208
2	773	1128	1666	2979	6276
3	1443	2587	4221	5995	7901
4	1174	1842	2711	3795	4538
5	584	1103	2971	4905	6525
6	715	1124	1896	4198	6235
7	2006	3414	5142	6644	8172
8	1295	2233	3107	4179	5193
9	1400	2383	3809	5381	7268
10	1009	1540	2200	3365	6642
11	729	1567	3993	5900	7466
12	827	1284	2127	3769	5157
13	1066	1763	2925	5187	7259
14	1146	1767	2500	4182	7030
15	1999	3874	6449	8025	9364
16	858	1641	3267	4316	5789
17	1361	2327	3246	4400	6554
18	881	1269	1925	2947	5442
19	1696	3106	4766	6030	7295
20	990	1518	2218	3231	4121
21	942	1917	3716	5212	6448
22	525	871	2059	5076	6670
23	1198	2899	4793	6665	8389
24	1528	2668	3599	4566	5630
25	1501	2692	4132	5353	6512
26	904	1378	1924	3294	7600
27	722	1671	4608	6828	8449
28	984	1561	2432	4460	5426
29	748	1182	2763	6170	7946
30	926	1387	2016	4515	7881
31	1564	3114	5408	7424	9031

Table 3.C.11 — Table for the first stage, lower part of the wideband VQ (lsp_tbl[1][Index][])

scale_factor = 2^15

Index	Codeword				
	0	1	2	3	4
0	7472	8934	10015	11175	13217
1	8233	9479	10725	12434	14154
2	7335	8745	10993	12330	13616
3	9920	11062	12168	13485	14666
4	6489	9325	10463	11799	13238
5	8528	9865	11277	12892	14502
6	7269	9858	11109	12483	14120
7	10116	11670	13053	14259	15501
8	6483	8177	9776	10950	12794
9	9462	10452	11420	12647	13795
10	7358	8940	9940	12541	13866
11	9136	10921	12617	14312	15907

12	5832	8075	11126	12420	13807
13	8475	10059	11736	13415	15033
14	6595	9134	10637	12879	14612
15	9831	11871	13624	15369	16869
16	7177	8241	10290	11834	12840
17	8737	9749	11079	12313	13420
18	8125	9201	10510	11858	13194
19	8977	10622	12235	13805	15243
20	6082	8849	10203	11627	14131
21	7740	9703	11425	13202	14809
22	7728	9142	10867	12826	14627
23	10868	12391	13674	14779	15831
24	6481	7886	8959	11538	13726
25	8873	10533	11925	13167	14460
26	7523	9139	10338	11718	14253
27	9226	11198	12943	14965	16628
28	6355	7787	10010	12579	13995
29	8071	10160	11937	14095	15765
30	5708	9623	11333	12828	14479
31	10737	12670	14292	15797	17066

Table 3.C.12 — Table for the second stage without interframe prediction, lower part of the wideband VQ (d_tbl[0][Index][])

scale_factor = 2^19

Index	Codeword				
	0	1	2	3	4
0	-2586	3968	2886	204	5966
1	1012	1572	-2381	4464	887
2	1167	1363	3292	853	1003
3	2042	3034	7092	3956	5132
4	4979	4760	-9300	-134	10831
5	-2905	-2776	674	2514	7577
6	3569	4039	2941	-3478	-1578
7	-1710	-548	3294	5376	3153
8	2186	3100	660	-1778	6145
9	-957	-795	3033	11341	-3589
10	4970	-4147	6490	4298	-572
11	417	7710	3470	273	-1453
12	-7	1046	-4507	2892	6555
13	-714	-1568	-2967	7020	4636
14	3772	3302	451	2446	-4140
15	2578	4384	9710	5358	-1217
16	-2532	-4202	7133	1880	5630
17	4804	5348	-3984	7228	3642
18	1111	3491	-1114	-430	1191
19	2678	3617	7802	-1392	-192
20	3806	8452	-2869	-3324	1352
21	3318	-6471	-1011	3546	7725
22	-3433	9091	3886	-6341	3227
23	2265	2960	3294	6694	-768
24	591	-253	91	-2479	9720
25	817	1668	4829	2207	-5378
26	6623	-2303	-3276	7908	1206
27	-4181	2006	12396	-134	3092
28	1687	1536	1396	4260	7281
29	-1745	-5444	996	9423	3266
30	6297	6104	5194	-160	-4582

31	-2136	-1987	7585	5756	-1961
32	738	1096	5525	-4589	4666
33	-645	-1470	265	5541	-1617
34	-2338	-431	7307	-347	-301
35	5627	6740	3288	2521	2080
36	10415	8027	-9823	-2155	3777
37	1325	-1970	-95	350	4293
38	3441	4597	5252	-8180	46
39	-6571	1403	-214	6608	4145
40	5454	3212	-1819	-6519	5840
41	1182	419	-1725	9004	-5253
42	7910	-2008	3452	-1016	4758
43	-3129	5539	2500	5557	-535
44	-349	9416	-3637	930	7559
45	1518	-2084	-9645	11648	4557
46	10192	415	-1341	1486	-5110
47	-2889	-1897	16428	3547	-5875
48	-3242	-265	6994	-2384	10971
49	5179	8334	-3396	4486	-3902
50	6665	2127	-3112	-489	1744
51	2773	6348	11843	-2666	-5595
52	8771	10848	112	-4867	-2993
53	-2051	-2805	-5007	3121	13573
54	4260	7545	5957	-4577	6355
55	1229	1851	2712	10833	3892
56	2501	4189	-323	-8917	13459
57	1756	3967	4998	7420	-8324
58	11279	-2442	-5259	2201	6640
59	-9527	9310	9443	2568	-1220
60	2819	5071	951	2904	13584
61	-3158	-4143	2986	14946	13769
62	-241	15823	5566	-2882	-8545
63	-4884	-7481	12990	9516	2628

Table 3.C.13 — Table for the second stage without interframe prediction, lower part of the wideband VQ (d_tbl[1][Index][])

scale_factor = 2^19

Index	Codeword				
	0	1	2	3	4
0	-2443	7932	-738	6162	-4136
1	-677	2914	7398	3994	-3822
2	-880	11860	48	-2227	3747
3	4653	-2433	-645	1289	5774
4	-5992	6932	4819	-4768	8843
5	4269	2134	4339	-913	-4016
6	263	2727	191	-150	822
7	3804	4965	330	3122	-3902
8	8235	-4048	447	3751	-2411
9	-10260	3203	8066	6308	472
10	33	2151	-404	6877	6237
11	6166	884	1015	-712	240
12	204	-3409	1364	5262	1605
13	1922	-2206	5780	510	1024
14	1655	4433	8847	-1457	4321
15	5754	7462	709	-1632	252
16	3275	830	-4011	5372	166
17	3562	7869	5134	71	-10033

18	-2276	8828	-7269	3063	4892
19	9444	-1435	-8008	3165	6199
10	-2110	3531	8135	5994	6521
21	11605	-5656	1614	-2073	3288
22	2976	3726	3134	3191	2662
23	10317	4135	-865	4701	3892
24	2399	469	2879	9433	-256
25	-2899	7059	5111	-1257	-1483
26	-643	-1379	-6800	4876	10284
27	3759	4372	-4707	714	6033
28	-3760	-4921	2991	3895	7671
29	71	-7924	6377	8260	1392
30	7179	3685	2958	-4074	7007
31	11439	2351	2992	-1922	-6670
32	2341	8850	-929	8357	1839
33	-2267	6105	13450	-523	-4823
34	1194	6805	-3370	-6589	7559
35	5727	-6273	160	4685	10448
36	-4980	-363	9224	-1874	4153
37	4819	-1241	10584	-1704	-4963
38	-3671	1662	2849	2825	1633
39	7957	1782	-753	8559	-7203
40	5991	-4262	7252	5705	-6894
41	-6883	5544	-1600	9315	2474
42	-1126	635	919	-748	7096
43	8914	1872	-3524	-5301	3577
44	-5824	-2532	1377	10418	2894
45	3915	-5883	8429	374	5885
46	2308	7252	4975	-7637	1080
47	4649	10765	-5746	1833	-2783
48	4233	407	-10181	10870	1626
49	6270	10391	8420	2016	-1259
50	-6408	5141	-729	1832	8457
51	5277	-5990	-2125	10813	4036
52	3332	5508	1806	2232	12391
53	7024	-1108	7624	-7499	1621
54	8546	280	6937	3987	1828
55	10174	1977	-4947	1058	-2864
56	-2769	315	4868	12245	-6828
57	-7240	11848	8060	3016	-3061
58	6405	-892	-5941	-3827	13649
59	9181	8383	-9470	-2700	6218
60	-5669	-1140	3997	10502	15094
61	-2045	-3486	12851	5447	-72
62	810	-2441	4093	-5399	11607
63	11810	9801	-1199	-7897	-3987

Table 3.C.14 — Table for the second stage with interframe prediction, lower part of the wideband VQ (pd_tbl[0][Index][])

scale_factor = 2^20

Index	Codeword				
	0	1	2	3	4
0	-423	8162	3591	1027	-1393
1	1510	-976	2127	-1862	5588
2	2712	1163	819	1966	-1098
3	1486	112	-2991	9568	-622
4	1310	1001	4839	7098	-5310

5	-1708	-4724	-1417	5607	9148
6	11270	12024	-9823	-2764	5319
7	-752	-1163	2297	5558	2639
8	3631	5562	10288	-3185	6849
9	-593	5589	3305	-8302	17200
10	6747	228	-4789	2016	3751
11	4539	5167	1948	9513	983
12	3071	22303	16697	3399	2876
13	2795	5989	1847	3168	6673
14	6878	6170	-824	2331	-7044
15	-2295	-6294	14996	1112	2109
16	2489	5774	5387	-8274	422
17	3293	6658	-3129	-4113	5510
18	664	18193	-504	9089	-5500
19	7492	6943	-7597	12275	-4567
20	2269	5895	7606	115	-7688
21	1370	1789	-6680	7488	7975
22	9523	12672	5301	-562	8445
23	1098	394	7784	-681	-368
24	-6107	6257	5386	-2910	5967
25	2687	3503	3902	2818	15370
26	7294	-7244	3126	1692	7991
27	55	447	153	16418	-8055
28	4039	4046	16811	2263	-6952
29	1351	1697	2436	10875	9000
30	5744	13856	-963	-4810	-3509
31	-4506	-709	5657	3139	7615
32	7301	2845	3243	-3558	-1997
33	-2526	-3081	6945	-4163	12184
34	2301	7126	-4694	3079	928
35	-1226	-3968	-3739	17350	7246
36	4379	5219	5428	9210	-12915
37	-1230	759	-3616	-536	11064
38	1094	8486	-12784	-556	15950
39	-2749	-4054	5048	12674	283
40	-2930	4167	14576	-11640	1847
41	4466	-561	-3958	-6857	19397
42	16820	3128	-3428	-6008	2652
43	2865	5115	8956	15768	-1583
44	-7399	25928	2084	-1257	3236
45	4036	4005	10262	6870	4940
46	11239	10153	6019	6280	-1579
47	-599	-6625	21393	13356	-4515
48	4022	12032	1243	-16378	5832
49	2807	2194	1544	-7964	8298
50	-7914	7848	-476	9582	1789
51	16219	2978	-4132	12303	5813
52	5665	8958	11103	-7796	-6245
53	3759	-5038	-9499	8712	16729
54	9512	16283	-1075	10111	12046
55	-5294	3076	10980	5078	-2573
56	-9705	8933	17367	5422	10101
57	-4006	-2796	-530	5854	18467
58	10783	-5541	5154	7238	-3177
59	2381	1238	2707	24266	2222
60	-8717	14128	16220	7269	-13329
61	2432	4218	6801	15890	16440
62	22723	16656	2318	-4151	-12759
63	-4697	-7096	9352	11102	11601

Table 3.C.15 — Table for the second stage with the interframe prediction, lower part of the wideband VQ (pd_tbl[1][Index][])

scale_factor = 2^20

Index	Codeword				
	0	1	2	3	4
0	9444	1828	-1379	664	1434
1	4316	-2360	7487	2691	232
2	-11548	800	13052	5765	8617
3	641	2689	-924	8929	-203
4	4737	814	2443	-6190	9333
5	-3842	-3076	909	7521	8486
6	1278	7620	3374	2521	-7664
7	-1781	4195	2105	-819	3093
8	15423	-3892	-9834	10664	448
9	9649	-731	3882	11712	7743
10	9704	3535	14839	4161	-9247
11	-5719	-2397	7548	9921	-2687
12	6872	11224	1951	-5952	-575
13	-11306	6686	-2318	8593	12745
14	-8832	10229	3631	5130	-156
15	2783	-624	-6826	3757	11469
16	8302	7502	-5167	6202	-7792
17	6088	-5881	94	12905	-2917
18	-113	-6158	15625	-3577	6335
19	2693	3359	-10442	15397	3406
20	13212	3987	-4853	-9899	13329
21	9749	-10490	2505	2257	9095
22	-1472	7684	11857	17536	-1232
23	1323	13374	-1514	-11289	12611
24	1977	-11506	-5158	16140	11268
25	13178	-4523	8932	-9732	5223
26	-3782	6838	11830	-3381	-598
27	-13884	-850	3082	23106	4970
28	19156	4585	235	-2255	-7658
29	4708	11415	1083	5521	11720
30	5705	12002	9256	5396	1247
31	953	10260	-6028	2010	1594
32	12552	-4987	5014	3963	-5039
33	6338	-10086	16190	8226	-4857
34	-8543	4389	3591	-3744	12922
35	-4168	13190	-3595	13328	-857
36	1596	6948	13535	-8044	11255
37	-2168	-13544	9163	11309	6468
38	-701	19627	8271	1945	-13273
39	473	3278	8118	4081	9567
40	14675	6431	-12422	276	4922
41	17014	5380	10270	1179	8012
42	7791	2301	9203	-6660	-3391
43	-6446	2897	19837	5364	-8208
44	16528	17154	767	-17143	-891
45	-12809	-2839	-419	10518	27038
46	-15472	21829	13893	7094	3026
47	-2340	10375	-10220	1851	13771
48	15023	9591	-166	12676	107
49	5362	-5174	6927	27166	-3086
50	6180	2204	18574	10349	9697

51	2408	8231	1457	21820	11880
52	1460	-408	-1465	-6386	27823
53	-427	-8411	6757	1010	18652
54	1356	5111	6358	15292	-14303
55	-2694	20171	5540	-3881	4060
56	11474	1094	-9006	9698	21887
57	23666	-7657	-1296	-414	5498
58	3539	11764	23025	-12228	-1961
59	-10130	-6581	24949	21344	10365
60	15803	16471	5757	352	-12586
61	-1231	13685	11266	7257	22928
62	7389	25109	10922	15030	2091
63	7499	21420	-7188	2244	1319

Table 3.C.16 — Table for the first stage, upper part of the wideband VQ (lsp_tbl[0][Index][])

scale_factor = 2^15

Index	Codeword				
	0	1	2	3	4
0	14931	17260	19135	20731	22466
1	16130	17759	19272	20469	21832
2	14608	16251	18746	20647	22478
3	16753	18362	20121	21770	23327
4	15552	16841	18257	20245	22266
5	16539	18104	19687	21273	22868
6	15101	16611	18049	19528	21357
7	17953	19286	20727	22128	23608
8	15756	17140	18998	20941	22707
9	16502	17762	18956	20549	22556
10	14473	15588	17705	20283	22335
11	17433	18714	19989	21305	22774
12	15968	17244	18508	19837	21410
13	15937	17691	19538	21274	22907
14	14948	15926	17038	19218	21851
15	18442	19887	21423	22906	24280

Table 3.C.17 — Table for the first stage, upper part of the wideband VQ (lsp_tbl[1][Index][])

scale_factor = 2^15

Index	Codeword				
	0	1	2	3	4
0	23968	25814	27338	28667	30395
1	24526	26064	27151	28531	29485
2	24511	26318	27711	29022	30694
3	23743	25259	27101	29078	30925
4	23698	25736	27594	29289	30968
5	23930	25559	26778	28291	29305
6	25175	26565	27871	29456	31049
7	22685	24448	26843	28939	30843
8	24480	25808	27105	28900	30888
9	25107	26620	27630	28864	29742
10	24549	26242	27882	29551	31152
11	22947	25325	27402	29152	30891
12	24232	25919	27608	29356	31064
13	23232	24879	26331	28014	29310
14	25418	27091	28522	29944	31227
15	23582	25139	26655	28449	30535

Table 3.C.18 — Table for the second stage without interframe prediction, upper part of the wideband VQ (d_tbl[0][Index][])

scale_factor = 2^19

Index	Codeword				
	0	1	2	3	4
0	-2560	2297	6975	798	-1926
1	-2735	3872	-4564	8120	5746
2	4373	947	-955	-914	1315
3	-908	4401	2667	-1287	1377
4	-2268	5853	10025	-6673	-163
5	1631	-28	2810	1904	-776
6	4776	6142	-2947	949	1415
7	-2833	-2024	-1112	4600	7474
8	-3864	2739	1593	4064	2570
9	6155	-1553	5820	4596	3997
10	9695	1441	-7817	1020	3792
11	-7730	2722	1634	1598	10322
12	3962	312	9630	2830	-2605
13	-2899	2110	2715	8973	-2241
14	12594	3871	-1360	-3011	-1569
15	-320	2494	3735	6371	9300
16	-757	3321	8918	3096	3193
17	2877	2783	2616	3357	3051
18	7251	-1824	3237	-1791	-634
19	-1369	9286	-1718	2204	129
20	4206	8370	4473	-668	4288
21	-2354	1560	6156	-856	8537
22	8254	4987	-2811	-8196	8943
23	3138	-3705	-698	2210	5013
24	-6427	-949	6278	3290	2970
25	8151	2254	-2434	6749	-5431
26	10346	-3913	-5767	8392	2997
27	924	1305	-2580	-1451	7512
28	2445	6006	2659	5765	-2794
29	-925	-3269	4200	6055	1733
30	5123	4744	7229	-3713	-3367
31	-6848	-4226	1631	11032	8429
32	-4902	2443	11540	4342	-6701
33	3104	1331	-2367	7153	1612
34	4133	2777	577	-437	-2305
35	2507	7589	2359	-615	-3543
36	-2694	9975	2067	-5305	6002
37	-984	-139	1242	1991	2742
38	6947	9379	-9670	-2294	2472
39	2133	-6472	35	10186	4359
40	-7413	6480	4407	2173	1195
41	6511	-5078	1092	4486	-556
42	11374	-5247	-2568	-352	6938
43	-631	8186	-5089	571	10441
44	8932	-2634	5515	968	-7191
45	-6319	-2454	6844	14208	63
46	9519	4144	3772	2319	-76
47	7155	-182	1678	11743	12984
48	1535	6830	7371	8848	3087
49	7643	2726	-1115	3775	7386
50	5540	1143	5004	-5812	4505

51	-1943	11191	6790	166	-1802
52	12402	12273	2874	-1611	2355
53	-976	-4154	8899	-646	3132
54	4645	7836	-96	-7035	547
55	387	-7025	3243	1364	9505
56	-12238	2499	6829	6815	5606
57	5210	-2761	4878	11656	-3536
58	3995	-5308	-7682	5622	15238
59	3851	777	-876	-3333	15812
60	3980	3863	6117	4229	-9652
61	408	-9600	9596	6683	2041
62	11040	9818	2056	-4210	-9529
63	-6364	-3140	3061	12611	19828

Table 3.C.19 — Table for the second stage without interframe prediction, upper part of the wideband VQ (d_tbl[1][Index][])

scale_factor = 2^19

Index	Codeword				
	0	1	2	3	4
0	1422	1178	933	494	245
1	2062	-50	1059	-1613	8136
2	6145	-3291	537	1902	636
3	5415	1583	3075	-34	-5337
4	3988	3443	4127	3416	2860
5	-1029	557	-693	4690	2872
6	5403	1770	-3950	1666	1720
7	-4164	3395	5114	1181	2743
8	571	6317	1269	-501	-1899
9	-1421	5826	-3067	449	6809
10	-194	-3079	5512	3052	1681
11	-66	233	2800	7851	-3319
12	6361	3473	-317	-4739	430
13	-4976	-1191	2889	10022	8119
14	3194	-4129	-3960	7152	6974
15	-3360	5872	10195	3113	-5435

Table 3.C.20 — Table for the second stage with interframe prediction, upper part of the wideband VQ (pd_tbl[0][Index][])

scale_factor = 2^20

Index	Codeword				
	0	1	2	3	4
0	-1203	944	1720	3038	3535
1	5458	5668	869	7066	-6563
2	3802	4199	2277	-1104	839
3	1430	-1000	-4780	1722	12432
4	-3481	4866	15357	-7220	5939
5	-39	-2956	9749	3445	1807
6	7477	17963	-7474	-6183	-826
7	4002	5024	-6877	3408	2368
8	-8652	7341	-6437	11935	8628
9	-10816	6463	16189	4581	485
10	17072	2029	7080	4366	7505
11	7739	8418	-3875	-10078	9607
12	10534	-3599	12877	-102	-3909

13	-7772	-6321	8296	3489	14360
14	14615	2173	-10507	590	4864
15	-1143	-5842	158	11118	3919
16	5949	5520	2089	7716	7633
17	-2745	-3749	6564	27658	945
18	11687	-4594	216	-6389	7029
19	11556	444	-13192	-4181	21192
20	-873	7995	10019	-2129	-5377
21	-369	-7926	20397	8882	-7803
22	607	15613	8257	12527	3173
23	4507	-3390	-12943	14817	7604
24	-4219	10154	2618	-1573	2973
25	-1579	1587	8487	10408	-3916
26	8905	-8575	-663	8312	4636
27	3444	17398	4070	2	10376
28	2063	7578	24687	4928	-5119
29	-245	3940	9563	14485	11203
30	12855	7088	695	-5315	-4689
31	-10581	-7648	-1577	16167	14056
32	-3105	10468	-1419	6311	-852
33	14162	4204	-7633	11235	-5516
34	9829	-1206	427	868	-594
35	7368	-7729	854	5349	19156
36	865	1591	5717	-6359	13827
37	5506	-9137	11190	-1984	8665
38	2610	21044	4984	294	-8907
39	5334	14543	-12802	6947	8133
40	-4716	10487	-8174	-1002	16152
41	-14240	1302	8133	16156	3859
42	14699	2379	8972	19069	3086
43	-2564	16674	3571	-14974	9552
44	12229	3721	8738	4077	-13926
45	-9855	6261	4855	1123	13212
46	24316	9325	-5634	-5552	5707
47	926	3508	-4145	18955	-1468
48	6134	7499	13623	3666	5145
49	8844	-7849	4340	12907	-5543
50	9855	6391	10506	-11434	1879
51	362	-983	-13635	6571	30326
52	3270	15666	19579	-10066	-10388
53	1502	-10344	15691	14109	6226
54	13715	15044	2230	6362	1724
55	15769	1574	-8019	16234	14189
56	-14732	20228	5382	1847	1919
57	-749	6117	11215	16353	-16089
58	23579	-7536	-1287	6310	-1700
59	14786	17464	1283	7296	20111
60	14265	20437	23019	9216	7590
61	712	7336	11183	11507	27807
62	25661	16635	5441	-2706	-6497
63	-492	-7793	-2651	31751	22163

Table 3.C.21 — Table for the second stage with interframe prediction, upper part of the wideband VQ (pd_tbl[1][Index][])

scale_factor = 2^20

Index	Codeword				
	0	1	2	3	4

0	6381	-7650	4834	7646	932
1	-4305	8028	-987	6332	2983
2	-5563	737	7443	2389	1608
3	1154	3384	5233	10481	-7363
4	14733	4933	2231	1558	-679
5	-3648	10342	3735	-4610	10841
6	71	-2562	-1188	4491	13847
7	4948	10669	3642	-4753	-2807
8	4454	-5104	9349	-5458	8376
9	2866	1850	1176	764	635
10	-7310	251	9250	17609	8502
11	-4875	17621	12117	3826	-3023
12	17912	-5764	-7648	-132	8401
13	6281	11385	-8672	-577	4955
14	4289	-2614	-7635	14525	4348
15	2977	1457	15505	5	-7012

Table 3.C.22 — Gain VQ tables for wideband CELP coder (subframe 5 ms)

scale_factor = 2^14

	Mode 0		Mode 1		Mode 2		Mode 3	
Index	Codeword		Codeword		Codeword		Codeword	
	0	1	0	1	0	1	0	1
0	599	1298	921	1797	1035	4241	3783	2563
1	3784	8406	4936	4301	7553	3257	10037	3498
2	581	7957	864	7173	2967	3207	6451	1634
3	4942	10321	3850	13075	10901	3097	11975	1865
4	1300	6724	2314	7483	633	9042	6235	3533
5	6920	7215	7528	7618	12395	6988	11098	3753
6	513	10942	4555	10181	6634	6057	9183	3336
7	1201	14544	12644	8988	16328	3625	14991	2517
8	2061	5070	2986	4312	3984	7127	5117	1456
9	3884	9804	7037	4080	9791	5902	10326	1756
10	2461	9152	1032	10962	6958	4985	8423	1702
11	4957	14317	7835	16139	12333	3345	13414	2265
12	3262	7068	5031	7939	2677	15607	6757	5391
13	7729	8838	10302	9184	14873	9446	12594	5533
14	2628	12220	6918	12161	8031	6654	10097	4761
15	5519	17797	10732	18523	20272	3969	18268	4229
16	578	4716	957	5617	3512	4340	2775	5249
17	4880	9642	6512	6810	9787	3094	10657	2882
18	583	9435	2089	8670	5721	2457	8336	2738
19	6077	11871	2134	16662	11813	5534	13085	3145
20	2025	8112	3683	9109	3648	9315	7153	3994
21	6078	10223	6776	10350	11161	9303	11618	3622
22	2063	11531	5198	11228	7688	9861	8988	5010
23	1356	21967	20897	8812	18483	7804	17195	5592
24	3273	5001	3963	6995	5995	4324	6806	3000
25	4036	10986	11311	4569	11124	5030	11677	2654
26	2718	10100	3197	11103	8130	3983	9377	1700
27	8055	16469	2497	29313	13682	6394	14588	4414
28	5235	6729	5934	8224	8616	14962	7609	7663
29	8609	11771	13311	13135	19242	11679	12796	6707
30	4352	12357	9255	12338	9999	8627	11878	4405
31	11833	21206	15036	28428	23752	7325	20724	3765
32	1790	4058	2329	3522	1095	6554	5370	3292
33	4381	8633	5799	5809	9046	3700	10525	3564
34	1587	9070	712	9386	5436	3537	7507	1546

35	5464	11198	5621	15640	11831	4584	12685	1954
36	2114	7225	3309	8079	1663	10700	5032	5233
37	6342	9273	8588	8857	12567	8603	11014	4705
38	1049	12492	5986	9462	7326	8023	9579	3823
39	3465	18246	15150	6736	18316	4608	16558	3172
40	2105	6254	4042	6080	5767	8387	5723	2219
41	4167	10407	9659	5438	9854	6755	11015	1839
42	3145	9183	1125	13013	8103	5217	9185	2491
43	7151	14299	5957	21510	15111	4637	13715	3115
44	3977	7482	4620	8888	2714	23350	8086	5592
45	8389	10349	11301	10090	16607	11483	13967	5190
46	3110	14265	6781	14406	8558	7235	10273	5808
47	7255	21228	15508	21007	25117	4387	21764	6654
48	1433	5237	2623	5611	4727	6014	3007	10841
49	5360	9203	7340	6003	9910	4686	11079	2581
50	1704	10340	2353	10036	6966	4041	8503	3675
51	6913	11744	2367	22911	13097	5213	13125	4655
52	2805	8173	3744	10150	5318	12239	7923	4231
53	7179	10672	7985	10213	12904	11892	12798	3945
54	3314	11295	5753	11999	8800	11173	9491	6396
55	3660	28526	21722	13636	20381	6825	18735	7428
56	3566	6378	4921	6642	5664	6616	7668	2787
57	4754	11395	10095	7628	10784	6714	12416	2964
58	3440	10239	4533	11966	8944	5064	10021	2579
59	13254	17336	8867	24609	15539	6973	14814	5391
60	5003	8520	6950	9494	14154	17320	10893	8085
61	11698	12315	14671	15807	18446	20359	15172	8048
62	5358	12793	10619	13654	10920	7946	11769	6293
63	13834	29363	23354	21673	24703	15306	29030	6036
64	1217	3385	970	3773	2118	5206	4362	4087
65	3884	9153	6109	4402	8589	2983	10093	4046
66	1233	8428	1157	8314	4418	3302	7110	2223
67	5514	10509	4160	14769	11371	3970	12164	2471
68	1446	7421	3142	7222	2312	8066	6436	4471
69	6677	8470	8489	7821	13225	7445	11390	4214
70	1339	11251	5135	10444	7225	6896	9699	3222
71	1586	17391	13607	10505	16576	5367	15330	3342
72	2617	5358	3718	4905	4923	7648	4838	2611
73	4400	9853	8193	4698	10381	5612	10563	2324
74	2540	9677	2080	11469	7508	5739	8802	2189
75	5796	15608	8973	17349	13874	3817	14129	2471
76	3361	7818	5350	8771	5746	16868	7281	6389
77	9229	9006	10300	11272	16274	8794	13319	5727
78	3283	12954	8018	11970	9116	6473	10556	5058
79	7833	18892	12609	19254	21761	5505	19430	5621
80	736	5974	1800	6476	4088	5205	4194	6743
81	5356	9852	6698	7888	10437	3967	10904	3276
82	1098	10008	2877	8999	6536	3172	8778	3014
83	6480	12689	3018	19568	12560	5915	13384	3764
84	2334	8655	4221	9541	5040	9895	7468	4847
85	6612	10298	7310	11011	11744	10640	12132	3724
86	2724	11412	6218	11104	8850	9212	9534	5349
87	3089	24397	18307	11421	18131	9618	16750	7371
88	4095	5433	4410	7552	6247	5217	7326	3299
89	4110	11619	12974	5577	11048	5901	11946	3120
90	2826	10652	3268	12117	8630	4487	9801	2108
91	10339	16358	7854	30218	14324	7749	15442	4383
92	5729	7590	6501	8819	11463	15415	8741	8959
93	9284	13013	15291	12012	21497	13238	13494	8009

94	4363	13274	9151	14161	10102	10112	12006	5155
95	12772	25369	20681	29050	24798	10061	24790	3779
96	2540	4393	1865	4907	3107	6252	5651	4337
97	4601	9188	6652	5401	9552	4135	10617	4209
98	1906	9616	1543	9909	4984	4537	7929	2192
99	6196	11030	5841	18197	12631	4421	12787	2581
100	2652	7549	4054	8309	2985	12035	5928	6327
101	7174	9672	9234	9576	13356	9540	11438	5050
102	2112	13108	5889	10320	8105	8329	9540	4472
103	4464	20881	16440	8650	18603	6196	16418	4537
104	2765	6423	4756	5557	6585	9426	6273	2626
105	4730	10792	9195	6994	10171	7491	11462	2165
106	3263	9696	2380	13715	8574	5861	9549	2705
107	8523	14450	8870	21111	14774	5944	14338	3508
108	4606	7810	5135	9562	7565	20550	8669	6361
109	9922	10880	11802	11997	17612	15318	14223	6533
110	3950	15507	7771	13519	9210	7896	11060	5833
111	7844	24825	18328	22318	28486	6690	24442	8629
112	1831	5906	3151	6223	5524	5478	5793	9717
113	5887	9674	8244	6383	10514	4767	11358	3086
114	2181	10586	3024	10188	7608	4508	9006	4056
115	7632	12635	4519	25658	13976	5215	13784	4333
116	3134	8648	4096	11021	7163	12105	8454	4613
117	7613	11318	8930	10905	14556	12567	12439	4499
118	3485	12027	6291	13058	10252	12315	10265	7043
119	7791	29772	27639	14102	21184	9147	19895	10141
120	4323	6777	5660	7239	6410	7279	7963	3426
121	5219	11927	11532	7212	11596	6692	12610	3439
122	3547	10753	5058	13115	9448	5297	10245	2992
123	16531	19141	11854	23948	16924	7041	15764	5995
124	5738	8568	7630	8963	12211	24436	11009	10871
125	11457	14382	17618	16423	23208	23197	15748	10756
126	5953	13714	11619	15720	11784	7975	12010	7860
127	20557	25673	27408	24006	29628	14761	28518	13371

Table 3.C.23 — Gain VQ tables for wideband CELP coder (subframe 2.5 ms)

scale_factor = 2^14

	Mode 0		Mode 1		Mode 2		Mode 3	
Index	Codeword		Codeword		Codeword		Codeword	
	0	1	0	1	0	1	0	1
0	623	1653	703	1600	925	1765	2183	1360
1	4977	7329	6999	7720	10136	2637	11461	1963
2	666	8305	773	4726	7284	2854	8235	1604
3	6899	10324	9384	5419	17693	3408	15241	1764
4	836	5878	5366	3996	3552	3626	6278	1420
5	8041	6466	1915	16812	13092	3610	13663	1633
6	758	10452	5218	9537	1252	8172	9952	1789
7	1744	17965	19240	5324	23891	4297	22255	2789
8	549	3524	3271	1975	930	4302	4727	1322
9	5259	9455	9741	10111	11475	7817	12848	3913
10	3357	7053	1016	7984	8913	5660	8032	5330
11	6477	14339	12019	10014	17203	6961	18901	2415
12	3465	4971	4268	6354	6085	3225	2755	4856
13	9498	11637	11057	19287	12780	7358	16607	5271
14	1162	13163	946	11254	1815	18430	9771	3962
15	14727	17847	18993	13700	21501	9618	24263	3548
16	2233	2742	2075	2655	2302	1585	3591	1370

17	6369	8148	9499	7235	10910	6438	12842	1755
18	2536	8587	2638	6053	8442	4614	8667	3083
19	7455	12035	12796	8078	20904	3754	16896	2250
20	1963	6182	7161	3209	4850	2132	7267	1518
21	9478	7743	2041	23208	15193	3309	14301	3564
22	3258	10620	6277	11342	1335	11752	11027	4360
23	2613	24379	24992	6118	30539	5101	21581	5803
24	2534	4510	3613	4377	5072	5039	5165	2753
25	4889	11162	7034	14112	10023	13225	14433	5337
26	3929	8618	3149	9271	4564	10549	2408	10035
27	10225	15484	14716	11416	17995	11869	18893	4618
28	5476	5282	5370	6841	7577	5740	7027	3594
29	12164	9762	15173	24543	14943	9463	18019	9866
30	3904	13407	4491	10826	14608	19852	10090	8292
31	16937	23095	29120	18723	25794	13415	28540	4549
32	719	2541	779	3070	2606	3061	3110	2077
33	5234	8582	8205	9287	11966	2871	12052	2613
34	1807	9290	859	6469	8410	2312	9039	1675
35	8211	9973	12740	5245	18666	5720	16124	2967
36	1498	7350	6175	5725	4664	3660	6560	2780
37	8017	8770	6764	16570	14048	4666	14753	2756
38	2227	10852	6582	9774	3404	7347	10781	2531
39	7023	18444	19832	10216	26317	4819	23634	5631
40	711	4714	4478	3309	3108	5525	5485	1551
41	5837	10768	10060	12511	13335	10213	13281	6175
42	4032	7748	2029	9324	10184	8061	9102	6383
43	8018	15511	12951	14163	19605	7670	20401	2633
44	4517	6184	4033	8206	6730	5149	5143	4680
45	12431	12041	14750	19319	14925	6376	17882	5722
46	1360	15172	1150	13784	2672	25528	10689	6099
47	21735	11850	21933	18982	25088	8345	25872	5708
48	3146	3669	2299	4178	3707	1495	4155	2796
49	6623	9260	11213	6767	12884	4993	13446	3316
50	2867	9706	2324	8065	10124	4932	10007	2862
51	9061	13290	14623	7410	21285	6916	17382	3230
52	2592	7688	7695	6137	5774	1902	7730	3213
53	10125	9396	5796	29402	16270	5653	15271	3473
54	3444	11804	7552	11915	5529	13556	11725	4997
55	9279	26510	25370	14528	28811	10300	23781	9389
56	2900	6049	4537	4853	5186	6805	5895	2900
57	6087	12216	9353	15797	10698	17281	15594	8576
58	4428	9889	3412	10687	8572	9470	4753	15856
59	9824	19814	15614	13667	21070	13055	20099	5645
60	6083	6650	5778	7953	7253	8188	6322	5956
61	14453	13532	19960	23752	13521	15297	18278	20753
62	4923	14800	4816	13435	19660	21913	12968	8969
63	24149	25073	26801	27969	26157	19536	29872	7509
64	1343	1958	1433	1972	1599	2840	1908	2712
65	5680	7722	7363	8708	10742	3638	12110	1597
66	845	9291	1692	5314	7933	3554	8326	2497
67	7852	10939	11043	4444	19097	3850	16095	1979
68	649	7044	6131	4275	4117	4580	6680	1974
69	8020	7694	4401	18565	14022	3151	14425	1890
70	1021	11649	5658	10466	3031	9278	10759	1586
71	4159	19643	21037	7928	24314	6271	22615	4297
72	1259	4010	3534	3208	1480	6154	4884	2127
73	5980	9889	10760	11043	12038	9471	13691	4463
74	4221	6962	791	9418	9774	6331	8909	4764
75	7905	13873	12597	11680	18157	8325	19421	3347

76	4371	5233	4516	7380	6633	3933	4270	5976
77	10680	12405	11586	22535	14104	7856	17412	4387
78	2586	13391	2157	12058	5314	19783	10044	4979
79	19007	19045	19934	16982	23597	10993	26254	3792
80	1980	3682	2644	3236	3174	2232	4178	1871
81	7128	8386	9236	8718	12090	5940	13302	2392
82	3176	8928	3308	6841	9449	4230	9172	3544
83	8469	12171	13966	9660	22246	5133	17895	2222
84	2427	6886	8753	3815	5250	2969	7755	2189
85	11337	7698	6214	23362	16354	3701	14966	4403
86	4069	10808	6373	12770	2903	14044	12013	3726
87	4726	29112	27001	8663	31096	8058	22199	7899
88	2566	5327	3626	5575	5905	5607	5405	3593
89	5277	11907	8489	13693	12120	12563	15585	5790
90	4537	9036	4155	9495	6506	10300	5333	8248
91	12014	17514	16827	10952	19001	15802	20115	4200
92	6832	5489	6418	6897	8057	6793	7221	4629
93	14981	10741	15658	29992	16610	10120	20225	11398
94	5273	13231	5049	11959	15893	27203	10868	11634
95	20157	28819	30803	23245	29259	15955	31248	4920
96	1319	2820	1504	3509	2374	4523	3315	3170
97	5901	8853	8577	10477	11889	4263	12697	2808
98	2006	10065	2016	6954	9045	3286	9333	2509
99	9150	10509	15152	4593	20107	5650	16662	3594
100	1775	8200	7212	4999	5640	4202	7201	2539
101	8961	9029	8028	19624	15144	4868	15429	2582
102	2385	11910	7351	10473	5328	8469	11381	3089
103	7377	22105	22295	12494	27918	6680	24775	7065
104	1734	5092	5147	2434	4266	6026	5886	2154
105	6678	11294	11181	13566	14762	12104	14331	7187
106	4702	8097	2271	10465	10433	10178	7955	9288
107	9089	17272	13586	16661	19612	10085	21153	3886
108	5263	6481	4910	8463	7461	4469	6051	4688
109	12274	14408	17041	20188	15871	7869	18924	6832
110	3309	15633	3010	14167	8007	29735	11824	6891
111	25017	18875	24828	19876	26724	9599	27400	7572
112	4402	3892	2859	4974	4059	2583	4363	3797
113	7410	9475	10800	8629	13674	6069	13961	2589
114	3733	9648	3238	7996	11064	5044	10569	3542
115	10146	13762	16439	8127	22951	7729	18249	3540
116	3305	8002	8169	7370	6582	2320	8174	4007
117	10831	10739	10658	28201	17431	5055	15913	4137
118	4326	12211	8946	11821	7740	15831	12650	5130
119	13330	28821	30514	12153	31223	12105	24856	13314
120	3668	6089	5168	5596	6491	6879	6295	3676
121	6832	12875	11177	16352	10034	22914	16956	7306
122	5036	10302	3554	12262	8204	11735	7943	19699
123	12133	22306	16900	16123	22923	16432	20369	7676
124	6891	7251	6187	8862	8955	7639	7342	6851
125	17114	15040	21821	28616	16435	14789	27140	20279
126	5790	16427	4909	15182	24024	27601	14456	13292
127	29293	29716	31068	29537	30256	23347	30549	11980

Table 3.C.24 — Filter coefficients for adaptive codebook interpolation (for 16 kHz sampling rate)

scale_factor = 2^15

Coefficients	Value
0	32767
1	31275

2	27042
3	20763
4	13437
5	6177
6	0
7	-4356
8	-6550
9	-6664
10	-5142
11	-2668
12	0
13	2202
14	3490
15	3703
16	2957
17	1578
18	0
19	-1357
20	-2185
21	-2349
22	-1896
23	-1020
24	0
25	890
26	1439
27	1552
28	1255
29	677
30	0
31	-588
32	-951
33	-1023
34	-825
35	-443
36	0
37	383
38	616
39	659
40	528
41	282
42	0
43	-239
44	-381
45	-404
46	-321
47	-169
48	0
49	142
50	224
51	236
52	186
53	98
54	0
55	-80
56	-126
57	-132
58	-105
59	-55
60	0

61	48
62	77
63	84
64	70
65	39
66	0

3.C.3 Gain tables for the bitrate scalable tool

Table 3.C.25 — Enhanced gain tables for bitrate scalable MPE tool (nge)

scale_factor = 2^14

Index	Mode 0	Mode 1	Mode 2	Mode 3
	Codeword	Codeword	Codeword	Codeword
0	460	3342	4827	9045
1	5049	3231	2242	1739
2	3328	1482	435	883
3	8745	5359	4010	4160
4	2155	135	1187	3582
5	6211	4113	3016	2736
6	4199	2452	1450	680
7	15087	8599	5760	7686
8	1199	1005	2490	5747
9	5562	3650	2617	2216
10	3775	2015	995	0
11	11112	6358	4691	5391
12	2832	848	241	2087
13	7228	4629	3477	3351
14	4615	2838	1854	1225
15	23253	15013	7969	11417

3.C.4 LSP VQ tables and gain VQ tables for the bandwidth scalable tool

Table 3.C.26 — LSP VQ tables for the bandwidth scalable tool (lsp_bws_tbl[0])

scale_factor = 2^13

Index	Codeword									
	0	1	2	3	4	5	6	7	8	9
0	-41	-58	-95	-284	-293	-417	-687	-734	-862	-1824
1	-78	-159	-181	-258	-225	-155	19	48	237	104
2	-68	-60	-140	-243	-236	-1003	-265	-489	-647	-792
3	-3	70	117	128	110	180	490	363	806	749
4	-50	-46	-74	-272	-225	-372	-497	-498	-1380	-1000
5	-4	46	59	46	82	17	111	23	66	629
6	-52	-87	-125	-234	-261	-408	-430	-358	-309	-165
7	34	144	228	389	413	363	921	581	1211	1604
8	-36	-57	-114	-255	-261	-520	-518	-1228	-501	-901
9	-6	56	104	117	156	194	198	367	191	78
10	-19	-39	-45	-110	-102	-168	-159	-267	-345	-579
11	43	228	387	391	422	601	644	1149	851	856
12	-28	-48	-70	-208	-222	-277	-396	-499	-563	-1032
13	23	118	249	214	470	357	396	664	266	785
14	-4	40	27	1	41	-17	8	2	-70	-177
15	29	181	406	504	654	829	1167	1610	1555	1759

Table 3.C.27 — LSP VQ tables for the bandwidth scalable tool (lsp_bws_tbl[1])

scale_factor = 2^13

Index	Codeword									
	0	1	2	3	4	5	6	7	8	9
0	10907	11766	12650	13565	14862	16624	18655	20941	22376	23585
1	11684	12595	13708	14900	16640	18214	19709	20667	21539	22525
2	11557	12621	13445	14584	15623	16358	17467	19933	22077	23331
3	12453	14102	15206	16263	17778	19035	20062	21238	22586	23799
4	10069	10772	12097	15128	17076	18536	20125	21164	22354	23609
5	11437	12290	14402	15497	16507	18259	20110	21510	22548	23866
6	10428	11550	13785	15584	17082	18505	19972	21362	22629	24040
7	13519	14437	15489	16474	17504	18223	19346	20260	21700	22671
8	11294	12217	13323	15370	16935	17819	19016	20873	22499	23702
9	12175	13269	14532	15637	16921	18040	19261	20229	21464	22346
10	12340	13277	14078	14979	15869	16793	19200	21345	22628	23859
11	13040	14209	15418	16742	17867	19029	20399	21350	22383	24253
12	10575	11293	11743	12973	15928	17866	19506	20907	22281	23775
13	12774	13427	14128	15005	16614	18510	20037	21408	22641	23998
14	11383	12366	13514	14730	16354	17611	18467	19590	21632	23199
15	14219	15088	15945	16742	17772	18822	20169	21096	22383	23076
16	11660	12545	13516	14524	15328	16601	19698	21415	22499	23376
17	12097	13307	14751	16156	17703	18738	19906	20757	21767	22627
18	11458	12811	14633	15588	16456	17350	18823	21007	22409	23592
19	12713	14672	15607	16709	17968	19510	20504	21681	22778	24083
20	10854	12003	13319	14259	16308	18370	19876	20997	22332	23913
21	11693	13666	14594	15448	17036	18421	19975	21474	22600	23992
22	11212	12873	14108	15210	17179	18468	19742	21081	22603	24048
23	13337	14268	15483	16817	18358	19295	20151	20896	21939	22724
24	11315	12649	13823	15061	17013	18564	21079	21928	22707	23308
25	12643	14086	15445	16473	17672	18695	19858	20768	21806	22616
26	13027	13721	14586	15957	17748	18844	19873	20745	21921	22785
27	13017	14220	15862	17090	18039	19147	20194	21511	22870	24079
28	11874	12624	13624	16234	17876	18853	19847	20702	21800	23113
29	12235	13632	15246	16266	17735	19063	20434	21858	22985	24491
30	11772	13540	15444	16547	17618	18357	19526	20418	21695	23442
31	14176	15233	16608	17791	18928	19646	20504	21269	22417	23112
32	11555	12468	13325	14186	16130	17528	18871	21143	22702	23983
33	12263	13275	14213	15361	16544	17688	19627	20733	22008	23127
34	12268	13260	14245	15156	16011	17040	18445	19590	21135	22502
35	12687	14062	15575	17189	18508	19599	20602	21568	22530	23829
36	11011	11731	12574	15253	17566	18568	19606	20538	21687	23153
37	12277	13236	14189	15629	17473	18858	20305	21733	22853	24353
38	10804	11753	13556	16294	18124	19224	20416	21279	22163	23291
39	13723	14568	15407	16425	17685	18831	20199	21009	22044	22759
40	12281	13269	14281	15760	17026	17823	19116	21239	22708	24006
41	13027	13935	14995	15941	16983	18139	19731	20719	21866	22601
42	12950	13976	14895	15815	16656	17439	18754	20665	22392	23447
43	13209	14239	15546	16814	18223	19333	20724	21848	22819	23543
44	11301	12020	12825	14070	17862	19418	20221	21125	22038	22948
45	12599	13411	14686	16302	17593	18998	20409	21827	22938	24441
46	11079	12232	14712	16017	17347	18206	19365	20374	21459	22870
47	14605	15535	16539	17338	18225	18959	19995	20885	22150	22882
48	11980	12961	13916	15070	17327	18500	19397	20507	22224	23689
49	12635	13608	14486	16291	17879	19395	20475	21406	22526	23907
50	12884	13800	14866	16695	17782	18659	19504	20597	22548	23877
51	13210	14328	15395	17103	18408	19454	20589	21687	22862	24303

52	11537	12323	13228	15167	17128	18898	20139	21372	22612	24073
53	11630	13730	14976	16399	17819	18939	20366	21637	22786	24166
54	11600	12937	14730	16090	17547	18924	20322	21772	22966	24445
55	14038	14990	16155	17219	18431	19453	20524	21208	22098	22768
56	12377	13483	14329	15311	17352	19158	20556	21374	22232	23017
57	13196	14174	15129	16082	17595	18849	20034	21032	22409	23146
58	13219	14289	14973	15687	16825	18633	20054	21364	22683	24070
59	13596	14613	15630	16771	18060	19138	20351	21647	22857	24294
60	11811	12929	14541	16053	17968	19782	20991	21708	22547	23559
61	12973	13958	15130	16734	18149	19201	20104	21750	22739	24049
62	11980	13767	15252	16959	17983	18928	20056	21157	22516	24032
63	14560	15534	16855	17972	19259	20100	21174	21854	22739	23280
64	11124	11915	13006	14838	15996	17157	19806	21653	22694	23721
65	11865	12936	14494	15839	17081	18236	19854	21004	22171	23000
66	12039	13102	13970	15107	16409	17327	18234	20445	22402	23722
67	12559	14111	15651	16731	17889	18967	20595	21688	22853	23943
68	10907	11714	12655	15055	17189	18884	20439	21654	22816	24213
69	12157	13171	14850	15851	16713	18603	20057	21359	22651	24068
70	11268	12146	14053	15983	17479	19054	20327	21583	22828	24213
71	13654	14819	16087	16993	17916	18623	19605	20522	22094	22989
72	11886	12696	13600	15575	16743	17762	19749	21490	22764	24053
73	12677	13744	15118	16219	17267	17986	19001	19888	21173	22142
74	12500	13587	14603	15564	16224	17338	19760	21668	23066	24193
75	13212	14183	15512	16436	17901	19524	20588	21663	22910	24196
76	11461	12178	12953	13883	15708	18666	20491	21381	22214	23141
77	12875	13648	14490	15698	17226	18289	20227	21858	23229	24271
78	11826	13288	14501	15868	17225	18048	18687	19656	22147	23559
79	14347	15334	16282	17307	18463	19509	20813	21608	22756	23358
80	12117	13159	14143	15321	16246	17682	20680	21755	22534	23266
81	12721	13508	15211	16121	17393	18360	19910	20824	22537	23381
82	11596	13812	15102	16106	17306	18166	19305	21120	22670	24020
83	12945	14571	15957	16990	18222	19348	20552	21777	23226	24419
84	11897	12862	14031	14917	16045	18237	19632	20995	22521	24045
85	12405	13813	14920	15812	17195	19245	20361	21635	22729	24025
86	10722	12804	14726	16072	17480	18862	20393	21364	22572	24020
87	13214	14506	16008	17050	18241	19275	20448	21361	22422	23196
88	12007	13171	14371	16289	17532	18732	21021	22062	22833	23501
89	13127	14043	15633	16744	17713	18819	20088	21178	22566	23242
90	12813	13993	15104	16112	17035	18061	20544	21715	22650	23342
91	13123	14793	15903	17038	18155	19189	20731	21785	22759	24035
92	12097	13086	14358	16610	18342	19402	20391	21148	22105	23037
93	12632	13900	15229	16543	17863	19170	20465	21966	22979	24529
94	11829	13441	15356	16337	17497	18756	20500	21414	22338	23098
95	14162	15325	16455	17659	18882	20060	21309	22418	23607	24472
96	12045	12847	13603	14450	16622	18426	20131	21616	22662	23924
97	12536	13617	14648	15575	16898	18591	19915	20970	22325	23043
98	12758	13636	14608	15798	16991	18225	19289	20383	21974	23815
99	12937	13718	15908	16869	18273	19405	20532	22671	23228	24035
100	11437	12259	13419	16195	18560	19580	20752	21828	23392	24403
101	12209	13019	15003	16484	17833	19177	20337	21436	22667	24075
102	11304	12537	14991	17148	18293	19139	20258	21199	22254	23559
103	13731	14744	15748	16970	18119	19390	20420	21416	22704	23536
104	12407	13326	14216	16176	17406	18491	19962	21156	22586	23850
105	13545	14580	15519	16456	17282	18164	19486	21053	22615	23852
106	12828	13959	15004	16154	17103	18199	18975	21215	23554	24506
107	13157	14462	15596	16821	18281	19763	20964	22012	23087	24329
108	11930	12741	13600	15086	17889	19365	20485	21511	22374	23246
109	13107	13994	14770	16370	17623	18984	20429	21574	23259	24282
110	11668	12599	14730	16358	17495	18417	19686	20984	22445	23896

111	15211	16134	17070	17965	18948	19620	20531	21290	22452	23034
112	12028	13013	13994	15115	16462	18398	20611	22476	23786	24570
113	12754	13834	15219	16509	18221	19449	20686	21448	22429	23138
114	12689	13824	15334	16647	17611	18465	19999	21569	22733	24159
115	13192	14300	16110	17226	18366	19584	20642	21645	22743	24179
116	11945	12739	13931	15701	17380	19109	20298	21508	22805	24138
117	12440	13777	15246	16751	18197	19637	20724	21662	22857	24211
118	11754	13477	14683	15761	17770	19081	20010	20890	22185	23886
119	13513	14663	15961	17337	18879	20081	21087	21778	22742	23409
120	12689	13744	14688	15645	17441	19107	21083	22232	23384	24243
121	13568	14439	15227	16313	17796	19363	20657	21617	22614	23544
122	13260	14617	15420	16243	17466	19002	20515	22061	23493	24525
123	13649	14770	15936	17149	18350	19477	20664	21918	23003	24228
124	12357	13518	14911	16697	18469	19951	21383	22319	23439	24360
125	13132	14203	15467	16836	18003	19226	20614	22038	23058	24481
126	12456	13734	15927	17116	18128	19069	20126	20945	21992	23021
127	15200	16311	17435	18492	19609	20520	21689	22532	23451	24216

Table 3.C.28 — LSP VQ tables for the bandwidth scalable tool (lsp_bws_tbl[2])

scale_factor = 2^13

Index	Codeword				
	0	1	2	3	4
0	-22	-5	-91	-439	36
1	-86	-180	-160	-22	116
2	19	-16	-10	-62	-383
3	-29	-32	67	-98	-32
4	29	54	-48	-89	222
5	-33	-132	61	158	-99
6	40	0	-121	106	-3
7	62	148	8	-85	-75
8	11	115	193	-72	122
9	-42	-15	64	127	160
10	26	89	129	154	-148
11	-18	66	432	178	101
12	-4	13	72	142	490
13	-21	-51	88	525	69
14	94	115	86	177	155
15	53	459	137	126	33

Table 3.C.29 — LSP VQ tables for the bandwidth scalable tool (lsp_bws_tbl[3])

scale_factor = 2^13

Index	Codeword				
	0	1	2	3	4
0	-274	-22	-435	135	-32
1	-61	-196	-282	-120	-75
2	-278	-398	105	-256	93
3	-398	-149	-392	-711	-473
4	-137	-85	-339	-351	285
5	14	-739	31	25	-160
6	-243	-109	-43	18	-169
7	-137	138	-257	-4	-204
8	-210	-25	-89	-202	61
9	494	285	-188	-275	-241
10	113	-400	-90	-301	115
11	-613	83	-80	-30	-12

12	29	-24	-65	-658	20
13	264	-111	-162	-295	-174
14	-319	-365	238	165	-236
15	-233	212	77	-100	-421
16	-180	-319	-126	142	235
17	128	-297	-472	131	-44
18	-25	-203	72	-219	-248
19	21	137	-129	-155	213
20	185	-113	-305	-84	304
21	69	-33	-280	-5	-338
22	-202	119	-86	298	-385
23	80	101	-460	-227	-42
24	96	132	-481	233	-113
25	139	123	-101	-183	-732
26	42	-191	-18	14	5
27	-235	160	111	3	244
28	-38	99	71	-4	-58
29	691	-152	-245	69	-42
30	6	-119	447	9	-247
31	-154	413	-234	-209	11
32	-76	-218	-204	228	-249
33	186	41	-114	-29	-75
34	-100	-155	358	13	171
35	-28	60	-105	-265	-216
36	-115	149	-268	142	400
37	242	-269	9	23	-347
38	-7	-63	54	60	-410
39	124	331	-116	57	-347
40	-30	-5	-248	103	105
41	73	213	123	-388	48
42	170	-430	143	74	371
43	-155	326	-95	239	40
44	-11	-87	95	-208	333
45	301	10	67	-183	236
46	91	-275	157	346	-81
47	-231	48	294	-269	-92
48	-19	-64	-291	588	29
49	255	-84	-178	278	150
50	185	-192	303	-272	9
51	-61	292	50	-279	685
52	61	20	40	136	317
53	220	-33	-150	317	-415
54	74	181	234	231	-329
55	221	283	-199	72	156
56	103	127	-40	275	-53
57	167	116	185	-192	-325
58	198	-43	197	106	35
59	-142	-23	14	292	85
60	120	270	198	2	127
61	405	132	99	108	-187
62	-198	121	359	243	-47
63	-107	511	244	-46	-107

Table 3.C.30 — LSP VQ tables for the bandwidth scalable tool (lsp_bws_tbl[4])

scale_factor = 2^13

Index	Codeword				
	0	1	2	3	4

0	97	-370	-55	31	-621
1	-692	315	193	-429	-730
2	-742	351	-448	-248	181
3	-223	-313	94	-619	-108
4	-104	-304	-774	-46	348
5	-415	-27	-405	-23	-113
6	-601	335	33	-3	-236
7	-970	-240	175	-132	-278
8	-408	-152	-46	-192	5
9	724	458	54	-493	-553
10	-395	233	-31	24	233
11	-404	-109	409	146	-609
12	207	-117	-364	-528	-616
13	208	141	0	-286	-420
14	-155	185	-441	167	-463
15	26	12	-389	616	-108
16	32	-171	-389	78	41
17	-346	491	-85	-545	164
18	-137	364	-33	228	-25
19	355	-270	-159	-337	-120
20	130	-186	-636	-1034	27
21	264	-381	-69	147	-126
22	-249	-826	454	256	-253
23	-417	-725	-608	630	138
24	-237	-267	-174	247	141
25	-373	147	338	-252	86
26	-124	20	86	-354	314
27	59	-468	176	-220	-95
28	400	181	-78	120	-714
29	196	347	109	15	-279
30	-1156	-675	408	435	264
31	193	-22	-75	285	72
32	96	-12	172	162	-329
33	-54	602	376	-129	-746
34	-309	419	-27	178	704
35	137	315	219	-197	43
36	62	-621	-232	21	249
37	269	51	-528	-253	76
38	-735	475	711	182	-296
39	-613	251	-212	474	14
40	13	-236	-57	-88	191
41	307	-44	517	121	-658
42	-269	-127	196	158	365
43	-169	-325	194	130	-208
44	185	126	-79	-571	39
45	462	389	51	213	292
46	-200	237	389	138	-80
47	145	-236	-599	669	814
48	593	-78	-32	530	-11
49	226	496	210	-440	442
50	-599	287	514	339	329
51	530	-83	-172	10	94
52	769	305	-85	-391	430
53	370	284	-179	364	-226
54	-48	-42	626	-103	314
55	-234	-206	58	856	432
56	106	197	136	126	90
57	31	-110	773	315	-201
58	167	7	264	-74	738

59	477	65	401	74	144
60	692	573	311	-12	-144
61	116	569	720	422	192
62	-174	-698	939	300	508
63	392	106	34	684	563
64	-374	-701	41	-235	-651
65	-440	28	-202	-527	-401
66	-730	-180	-223	156	322
67	153	-242	252	-389	-633
68	-726	-567	-587	-558	240
69	-89	-380	-679	-263	-476
70	-1226	531	195	-86	93
71	-539	-403	-120	292	-259
72	-201	-476	-212	-102	-164
73	259	161	-71	-674	-1205
74	-665	-14	118	-378	563
75	-331	-306	598	-210	-287
76	-732	-870	-1259	-2795	-3315
77	-14	-120	-172	-200	-235
78	-298	29	-22	-39	-538
79	-82	-425	-477	375	-342
80	394	-264	-699	175	-118
81	-121	164	-398	-408	-69
82	-114	319	-416	106	144
83	-45	-292	-298	-461	142
84	-24	71	-321	-729	644
85	-134	-87	-164	219	-216
86	-367	-553	241	-152	187
87	-28	-514	161	408	103
88	-498	-80	210	153	-15
89	-180	245	-34	-210	-91
90	146	-406	236	-472	418
91	350	-373	457	24	-188
92	278	-80	-348	81	-450
93	-71	455	-196	-147	-422
94	-451	-334	642	569	-19
95	46	25	-124	206	416
96	-148	-13	225	-206	-260
97	-87	547	244	-440	-279
98	339	246	-263	-78	816
99	345	-17	333	-422	7
100	287	-316	180	198	338
101	133	170	-253	32	-98
102	-356	826	285	4	70
103	-39	433	17	563	234
104	-252	-39	-321	-190	378
105	620	-52	99	19	-331
106	-198	-408	-72	-1	702
107	158	-297	113	509	-376
108	583	241	-406	-798	-85
109	86	268	-177	-184	284
110	-228	228	72	417	-390
111	-309	73	-463	455	418
112	193	-359	-363	643	229
113	123	640	-113	-54	101
114	-60	363	268	16	386
115	241	-18	57	-68	-77
116	359	-218	-373	-235	507
117	-157	-18	101	482	25

118	24	-142	308	26	71
119	75	-2	382	375	269
120	-144	3	-10	16	18
121	125	200	545	-80	-266
122	288	20	51	-153	329
123	299	127	282	424	-132
124	481	271	-173	-188	-100
125	66	600	367	296	-318
126	247	-286	555	781	101
127	453	136	-466	306	279

Table 3.C.31 — LSP VQ tables for the bandwidth scalable tool (lsp_bws_tbl[5])

scale_factor = 2^13

Index	Codeword				
	0	1	2	3	4
0	-231	-647	-207	108	-41
1	8	-228	-481	-388	60
2	182	-257	245	105	-240
3	-383	-213	-24	-290	-364
4	-494	-67	-22	106	177
5	238	149	-131	-415	-434
6	-130	370	-149	-166	239
7	-294	344	288	-86	-347
8	92	-234	-394	307	318
9	546	22	-210	-92	125
10	5	46	-196	176	-154
11	25	-93	187	-260	199
12	-128	-329	318	349	351
13	422	469	252	-33	-133
14	238	174	114	339	316
15	-172	311	472	307	46

Table 3.C.32 — LSP VQ tables for the bandwidth scalable tool (bws_ma_prdct)

scale_factor = 2^15

	Order		
Codeword	0	1	2
0	32307	6025	3826
1	29035	5743	3609
2	32346	4386	2552
3	29547	5161	2645
4	27530	6708	4095
5	28237	8367	4965
6	27312	8564	4803
7	24644	8340	4905
8	27090	10058	5834
9	23277	9490	5693
10	20474	8283	4014
11	21357	7799	3607
12	22134	7214	3424
13	22373	6960	3431
14	21710	7190	3887
15	20912	7440	4427
16	20712	7440	4634
17	20797	7242	4759
18	20987	6867	4921

19	23026	6144	3612

Table 3.C.33 — LSP VQ tables for the bandwidth scalable tool (bws_nw_prdct)

scale_factor = 2^15

Codeword	Order
0	16438
1	16363
2	16258
3	16625
4	16492
5	16832
6	16688
7	16911
8	16822
9	17241

Table 3.C.34 — Gain VQ tables for bandwidth extension tool

scale_factor = 2^13 for Codeword 0, 2^10 for Codeword 1

	Mode 0		Mode 1		Mode 2		Mode 3	
Index	Codeword		Codeword		Codeword		Codeword	
	0	1	0	1	0	1	0	1
0	29	10	72	16	36	4	22	5
1	155	1313	1527	428	3773	164	4836	118
2	139	691	66	470	2280	147	3203	107
3	4311	1706	3693	442	184	789	4417	258
4	52	453	654	407	232	429	2521	193
5	7966	1055	2237	548	5995	164	6309	166
6	437	1062	1102	572	4182	262	3493	134
7	1482	4291	214	884	2085	2647	10201	139
8	267	245	431	290	503	214	1577	128
9	1721	1509	2608	456	4927	167	5443	115
10	3405	424	311	573	2878	157	3365	287
11	1202	2863	4799	447	1005	1218	5688	222
12	2032	319	1555	501	2604	430	3410	545
13	6916	1810	3188	651	6883	193	7856	112
14	3862	990	2265	602	4740	540	4120	104
15	3565	7136	3853	1489	371	6591	12458	88
16	332	72	288	141	428	57	591	312
17	335	2036	2305	494	4755	255	5426	186
18	2385	725	448	490	2389	273	2962	204
19	4815	2630	3789	598	493	956	4896	349
20	1120	468	886	510	1678	392	2300	461
21	6648	1551	2753	570	6457	382	6572	202
22	2272	999	1365	636	4938	405	4062	211
23	1474	5520	2073	903	4991	3408	11982	58
24	355	341	1800	274	1692	161	2375	89
25	3714	1293	3019	522	5496	256	5844	122
26	3851	738	332	689	3285	269	3997	341
27	9201	3462	6836	583	4511	1297	5709	313
28	2015	515	1746	553	3331	447	5663	735
29	12655	2009	3902	684	8027	246	7292	390
30	5839	949	1904	752	6078	868	4790	167
31	4604	13921	1932	2359	7119	16128	8754	232
32	326	36	188	84	152	32	131	135
33	628	1564	2057	427	4486	161	5023	164

34	1278	660	453	597	2706	233	3139	156
35	3767	2135	3900	560	596	855	4638	295
36	26	568	1068	447	982	281	2449	314
37	10488	1524	2500	574	6249	291	7008	161
38	828	949	1469	581	4458	377	3655	182
39	9425	4374	579	1221	8901	2654	11389	95
40	1130	264	948	316	1261	213	2040	131
41	2480	1676	3062	465	5376	160	5664	182
42	3932	542	767	621	3666	230	3805	265
43	4684	3644	4726	600	283	1452	6126	294
44	2514	389	1924	503	2226	616	4259	1073
45	7858	2378	2984	685	7234	309	7348	263
46	3288	1131	2576	645	6374	688	4494	116
47	1949	9019	9106	1394	21348	8971	13310	134
48	244	158	757	179	636	114	1699	319
49	1465	2347	2542	508	5070	285	5198	272
50	2440	860	715	546	2894	324	3297	225
51	5127	3083	4130	622	2433	977	5217	424
52	1409	548	1327	533	845	570	3136	424
53	9384	1804	2884	620	7343	531	6585	275
54	1700	1179	1845	645	5696	534	4361	169
55	14524	5593	2195	1168	2014	4582	11911	95
56	949	352	2252	363	1713	239	2886	97
57	5212	1499	3302	571	5775	319	6061	201
58	5362	700	1089	710	3690	354	4434	372
59	14442	3913	7643	1058	9267	1586	5986	489
60	2555	625	1883	593	3502	555	8981	919
61	11609	2905	4804	809	9577	442	7210	513
62	5427	1277	2983	771	6900	1105	5013	231
63	0	31787	5778	4721	22789	22049	9665	362
64	89	29	290	46	71	16	316	37
65	1238	1345	1710	468	4123	199	5119	119
66	697	792	21	534	2438	200	3639	79
67	4938	1937	4056	507	368	858	4687	236
68	645	496	795	472	237	513	2681	242
69	8986	1283	2267	575	6402	207	6626	137
70	259	1138	1178	603	4408	299	3826	136
71	4997	4849	905	969	3979	2327	11173	47
72	682	279	367	359	443	271	1777	208
73	3340	1543	2820	495	5127	220	5597	147
74	3153	528	332	619	3270	195	3587	338
75	1141	3398	5130	537	2115	1347	5898	255
76	1869	409	1713	518	2826	517	4038	639
77	7180	2034	3458	655	7379	224	8178	158
78	4736	998	2287	639	5138	669	4157	148
79	14840	7980	6170	1582	11237	7536	12675	100
80	680	106	344	215	178	109	1118	505
81	1687	2022	2401	529	4776	322	5365	228
82	3014	779	558	529	2431	342	3013	265
83	7087	2675	3723	635	114	1132	5295	332
84	1651	469	1090	530	2112	424	2150	714
85	7959	1618	3005	570	7028	406	6909	226
86	2427	1091	1535	670	5424	426	4303	215
87	6492	5822	2768	1003	7010	3938	12193	55
88	656	406	2464	234	2067	216	2601	123
89	4191	1420	3253	538	5953	240	6124	134
90	4479	834	559	767	3484	305	4300	315
91	13038	3429	6516	773	5118	1514	5794	382
92	2478	543	1976	566	3806	441	7203	762

93	14216	2430	4054	748	8929	295	7748	341
94	6018	1114	2372	775	7857	891	4868	202
95	3475	20505	5337	2943	3759	14544	8590	358
96	397	42	796	97	344	28	1165	191
97	382	1663	2101	473	4510	220	5287	153
98	1702	773	325	651	2964	253	3369	179
99	5845	2257	4305	566	1340	816	4914	278
100	829	587	1305	483	981	371	2863	340
101	13478	1351	2613	607	6686	292	7459	185
102	1678	933	1657	612	4322	466	3916	182
103	10888	4993	207	1579	12791	3031	11089	182
104	1585	218	1274	360	1448	299	2276	167
105	2991	1866	3427	501	5609	196	5898	175
106	4919	539	1015	651	3868	281	4101	269
107	8233	3965	5274	687	1575	1684	6346	355
108	2585	462	2070	527	2962	713	4180	2808
109	10285	2523	3345	729	8010	388	7974	250
110	4196	1172	2521	690	4466	935	4597	151
111	16560	10713	11683	1915	8917	11836	13208	206
112	791	196	1199	220	1020	154	2152	248
113	3636	2419	2675	539	5337	334	5511	269
114	3207	912	814	573	3214	373	3602	229
115	8462	3005	4503	673	2528	1076	5061	562
116	1899	621	1506	549	1531	520	3877	422
117	9930	2086	3203	606	8439	576	6895	323
118	2647	1322	2063	689	6391	498	4560	199
119	12868	6493	4074	1172	10597	4788	12074	108
120	1286	408	3429	341	1963	301	2836	155
121	5676	1686	3525	577	5940	399	6269	232
122	7086	803	1352	804	4069	361	4552	459
123	14891	4520	10736	841	6723	1888	6633	430
124	3296	639	2029	620	3979	632	12932	1373
125	15411	3038	5187	987	11638	607	8281	562
126	7025	1326	3450	879	9588	1258	5173	197
127	1466	336	8587	10865	0	26530	11167	457

Table 3.C.35 — Gain tables for excitation codebook 1 of bandwidth extension tool

scale_factor = 2^12

Index	Mode 0	Mode 1	Mode 2	Mode 3
	Codeword	Codeword	Codeword	Codeword
0	-9890	-334	-1119	-6181
1	-3715	429	-55	-2339
2	-1164	1003	557	-996
3	180	1442	1137	-261
4	1124	1802	1625	243
5	1717	2116	2059	708
6	2378	2403	2470	1157
7	3013	2684	2915	1641
8	3519	3001	3408	2180
9	4213	3349	3914	2802
10	5112	3750	4541	3361
11	6184	4275	5297	4052
12	8150	5030	6450	5008
13	11196	6491	8837	6691
14	16806	9366	14863	10432
15	26264	27589	31529	22189

Annex 3.D
(informative)

Tables

3.D.1 Bandwidth expansion tables in LPC analysis of the mode II coder

Table 3.D.1 — Bandwidth exapnsion tables in LPC analysis

For 8 kHz sampling rate	
0	1.0001000
1	0.9988903
2	0.9955685
3	0.9900568
4	0.9823916
5	0.9726235
6	0.9608164
7	0.9470474
8	0.9314049
9	0.9139889
10	0.8949091

For 16 kHz sampling rate	
0	1.0000100
1	0.9997225
2	0.9988903
3	0.9975049
4	0.9955685
5	0.9930844
6	0.9900568
7	0.9864905
8	0.9823916
9	0.9777667
10	0.9726235
11	0.9669703
12	0.9608164
13	0.9541719
14	0.9470474
15	0.9394543
16	0.9314049
17	0.9229120
18	0.9139889
19	0.9046499
20	0.8949091

3.D.2 Downsampling filter coefficients for the bandwidth scalable tool

Table 3.D.2 — Downsampling filter coefficients

0	0.487498	25	0.006314	50	0.003645	75	0.001263
1	0.318007	26	0.009227	51	0.001586	76	-0.000190
2	0.012479	27	-0.005043	52	-0.003242	77	-0.001148
3	-0.105197	28	-0.008776	53	-0.001706	78	0.000159
4	-0.012414	29	0.003938	54	0.002858	79	0.005156
5	0.062159	30	0.008312	55	0.001785	80	-0.000092
6	0.012306	31	-0.002977	56	-0.002497		
7	-0.043381	32	-0.007836	57	-0.001830		
8	-0.012155	33	0.002141	58	0.002158		
9	0.032701	34	0.007355	59	0.001844		
10	0.011965	35	-0.001414	60	-0.001842		
11	-0.025712	36	-0.006869	61	-0.001831		
12	-0.011735	37	0.000785	62	0.001551		
13	0.020722	38	0.006383	63	0.001794		
14	0.011468	39	-0.000241	64	-0.001284		

15	-0.016942	40	-0.005902	65	-0.001737
16	-0.011167	41	-0.000222	66	0.001042
17	0.013954	42	0.005428	67	0.001663
18	0.010830	43	0.000616	68	-0.000825
19	-0.011527	44	-0.004961	69	-0.001576
20	-0.010467	45	-0.000942	70	0.000632
21	0.009494	46	0.004508	71	0.001478
22	1.01E-02	47	0.001210	72	-0.000462
23	-7.78E-03	48	-0.004068	73	-0.001373
24	-9.66E-03	49	-0.001423	74	0.000315

Annex 3.E
(informative)

Example of a simple CELP transport stream

Two types of streams, header streams and raw data streams, are part of the MPEG-4 CELP coder syntax. The header streams contain configuration information necessary for the decoding process and parsing of the raw data streams. The raw data streams contain the coded audio information. A transport layer handles the delivery of these streams. The multiplexed bitstream of the header and the raw data stream is shown in the following pseudo code example:

```
CelpHeader()
if (BandwidthScalabilityMode == ON)
{
   CelpBWSenhHeader();
}
while(1)
{
   CelpBaseFrame();
   for (i = 0; i < nrof_enh_layers; i++)
   {
      CelpBRSenhFrame();
   }
   if (BandwidthScalabilityMode == ON)
   {
      CelpBWSenhFrame();
   }
}
```

The header information is attached at the begining of the bitstream. Its update is necessary only when changes are made in the configuration of the bitstream. The raw data information follows the header stream and is transmitted frame by frame. In a scalable bitstream, two types of information, namely base information and enhancement information, are transmitted every frame.

In order to control transmission of the enhancement information, which is in CelpBrsenhFrame() or CelpBwsenhFrame(), frame by frame, the decoder requires additional parameters which indicate whether the enhancement information is available for decoding at every frame. The parameters for bitrate scalability and bandwidth scalability are *dec_enh_layers* and *dec_bws_mode*, respectively. The following procedure is applyed to parse the bitstream.

```
CelpHeader()
if (BandwidthScalabilityMode == ON)
{
   CelpBWSenhHeader();
}
while(1)
{
   CelpBaseFrame();
   for (i = 0; i < dec_enh_layers; i++ )
   {
      CelpBRSenhFrame();
   }
   if (BandwidthScalabilityMode == ON && dec_bws_mode == ON)
   {
      CelpBWSenhFrame();
```

```
    }
}
```

where *dec_enh_layers* is the number of available enhancement layers in the curent frame and is equal to or less than the value of *nrof_enh_layers* in the header information. *dec_bws_mode* is a one-bit parameter that indicates whether bandwidth scalability is enabled in the current frame as follows:

Table 3.E.1 — Definition of dec_bws_mode

dec_bws_mode	ScalableID	Description
0	OFF	Bandwidth scalability is disabled
1	ON	Bandwidth scalability is enabled

Annex 3.F
(informative)

Random access points

Random access to the CELP bitstream is available at every frame. However, the decoder should be initialised before decoding the first frame. If FineRate Control is enabled the LPC filter coefficients should be initialized to zero as no LPC parameters might be present in the first frame being parsed.

Contents for Subpart 4

Subpart 4: General Audio Coding (GA) – AAC, TwinVQ, BSAC

4.1 Scope

The General Audio (GA) coding subpart of MPEG-4 Audio is mainly intended to be used for generic audio coding at all but the lowest bitrates. Typically, GA encoding is used for complex music material in mono from 6 kbit/s per channel and for stereo signals from 12 kbit/s per stereo signal up to broadcast quality audio at 64 kbit/s or more per channel. MPEG-4 coded material can be represented either by a single set of data, like in MPEG-1 and MPEG-2 Audio, or by several subsets which allow the decoding at different quality levels, depending on the number of subsets being available at the decoder side (bitrate scalability).

MPEG-2 Advanced Audio Coding (AAC) syntax (including support for multi-channel audio) is fully supported by MPEG-4 Audio GA coding. All the features and possibilities of the MPEG-2 AAC standard also apply to MPEG-4. AAC has been tested to allow for ITU-R 'indistinguishable' quality according to [4] at data rates of 320 kb/s for five full-bandwidth channel audio signals. In MPEG-4 the tools derived from MPEG-2 AAC are available together with other MPEG-4 GA coding tools which provide additional functionalities, like bit rate scalability and improved coding efficiency at very low bit rates. Bit rate scalability is either achieved with only GA coding tools, or by using a combination with an external (non-GA, e.g. CELP) core coder.

MPEG-4 GA coding is not restricted to some fixed bitrates but supports a wide range of bitrates and variable rate coding. While efficient mono, stereo and multi-channel coding is possible using extended, MPEG-2 AAC derived tools, the document also provides extensions to this tool set which allow mono/stereo scalability, where a mono signal can be extracted by decoding only subsets of the encoded stereo stream.

4.1.1 Technical Overview

4.1.1.1 Encoder and decoder block diagrams

The block diagrams of the GA encoder and decoder reflect the structure of MPEG-4 GA coding. In general, there are the MPEG-2 AAC related tools with MPEG-4 add-ons for some of them and the tools related to the TwinVQ quantization and coding. The TwinVQ is an alternative module for the AAC-type quantization and it is based on an interleaved vector quantization and LPC (Linear Predictive Coding) spectral estimation. It operates from 6 kbit/s/ch and is recommended to be used below 16 kbit/s/ch with constant bitrate.

The basic structure of the MPEG-4 GA system is shown in Figure 4.1 and Figure 4.2. The data flow in this diagram is from left to right, top to bottom. The functions of the decoder are to find the description of the quantized audio spectra in the bitstream payload, decode the quantized values and other reconstruction information, reconstruct the quantized spectra, process the reconstructed spectra through whatever tools are active in the bitstream payload in order to arrive at the actual signal spectra as described by the input bitstream payload, and finally convert the frequency domain spectra to the time domain, with or without an optional gain control tool. Following the initial reconstruction and scaling of the spectrum reconstruction, there are many optional tools that modify one or more of the spectra in order to provide more efficient coding. For each of the optional tools that operate in the spectral domain, the option to "pass through" is retained, and in all cases where a spectral operation is omitted, the spectra at its input are passed directly through the tool without modification.

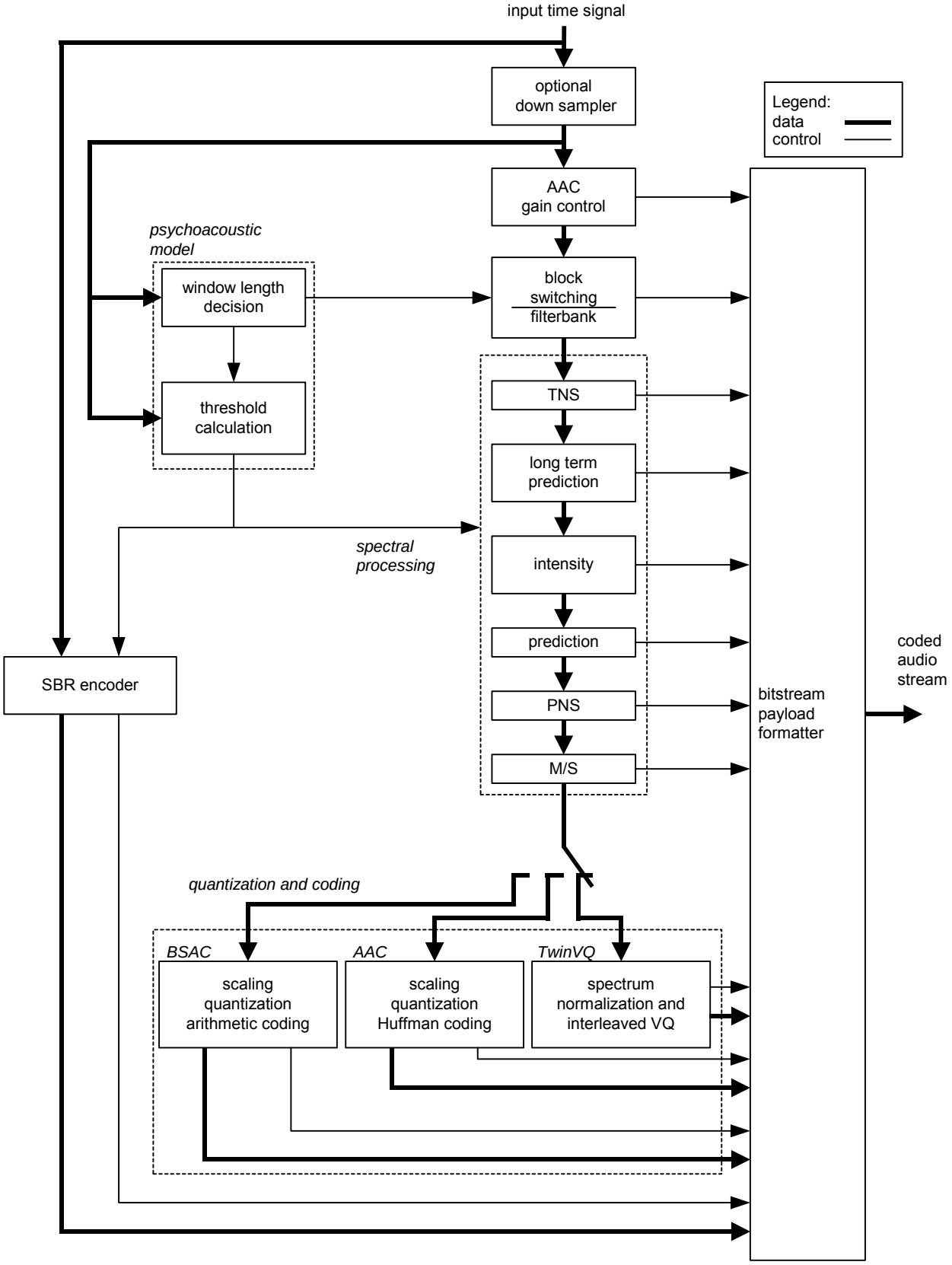


Figure 4.1 – Block diagram GA non scalable encoder

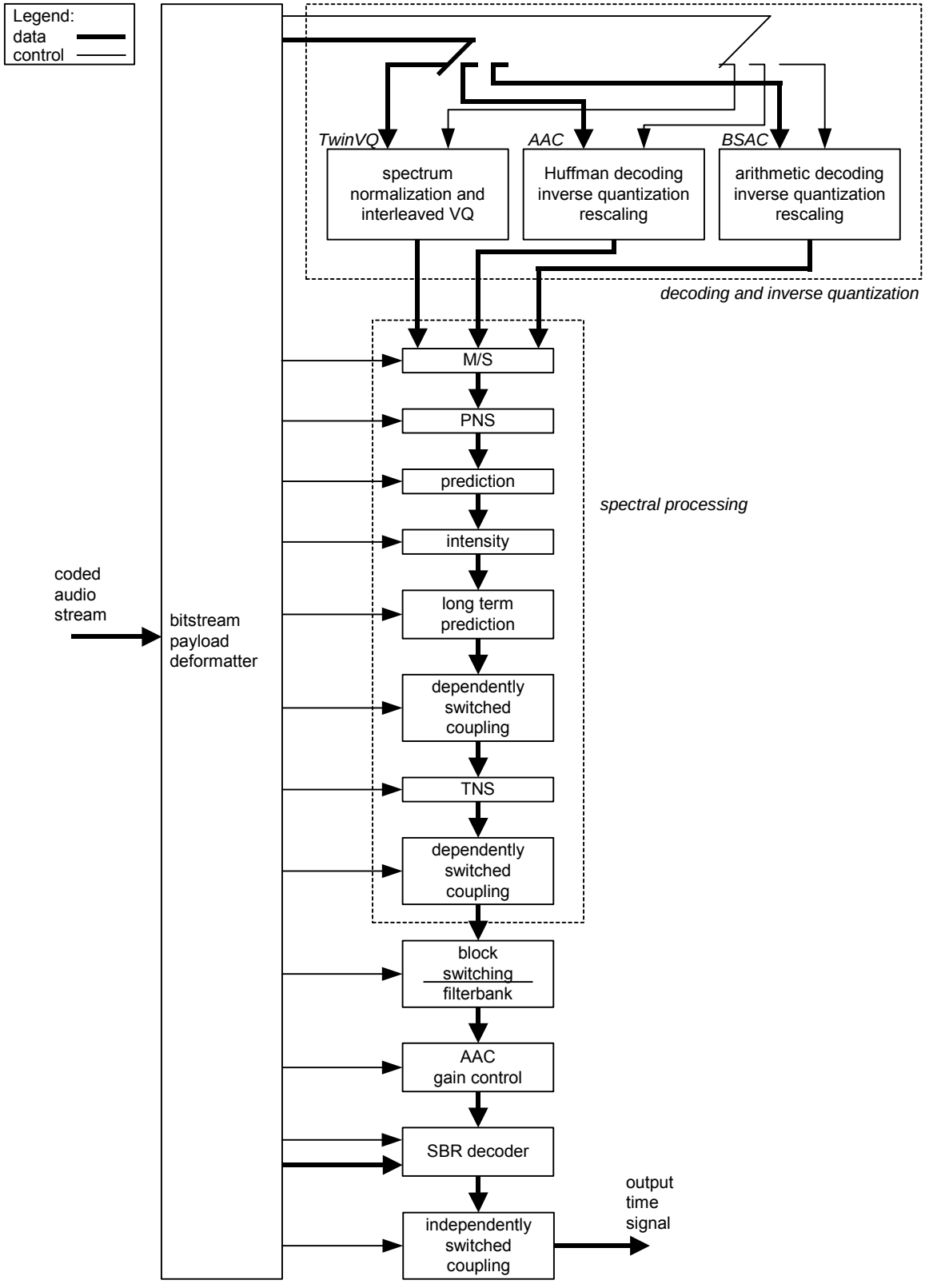


Figure 4.2 – Block diagram of the GA non scalable decoder

4.1.1.2 Overview of the encoder and decoder tools

The input to the bitstream payload demultiplexer tool is the MPEG-4 GA bitstream payload. The demultiplexer separates the bitstream payload into the parts for each tool, and provides each of the tools with the bitstream payload information related to that tool.

The outputs from the bitstream payload demultiplexer tool are:

- The quantized (and optionally noiselessly coded) spectra represented by either
  - the sectioning information and the noiselessly coded spectra (AAC) or
  - a set of indices of code vectors (TwinVQ) or
  - the arithmetic model information and the noiselessly coded spectra (BSAC)
- The M/S decision information (optional)
- The predictor side information (optional)
- The perceptual noise substitution (PNS) information (optional)
- The intensity stereo control information and coupling channel control information (both optional)
- The temporal noise shaping (TNS) information (optional)
- The filterbank control information
- The gain control information (optional)
- Bitrate scalability related side information (optional)

The AAC noiseless decoding tool takes information from the bitstream payload demultiplexer, parses that information, decodes the Huffman coded data, and reconstructs the quantized spectra and the Huffman and DPCM coded scalefactors.

The inputs to the noiseless decoding tool are:

- The sectioning information for the noiselessly coded spectra
- The noiselessly coded spectra

The outputs of the noiseless decoding tool are:

- The decoded integer representation of the scalefactors:
- The quantized values for the spectra

The inverse quantizer tool takes the quantized values for the spectra, and converts the integer values to the non-scaled, reconstructed spectra. This quantizer is a non-uniform quantizer.

The input to the Inverse Quantizer tool is:

- The quantized values for the spectra

The output of the inverse quantizer tool is:

- The un-scaled, inversely quantized spectra

The rescaling tool converts the integer representation of the scalefactors to the actual values, and multiplies the un-scaled inversely quantized spectra by the relevant scalefactors.

The inputs to the scalefactors tool are:

- The decoded integer representation of the scalefactors
- The un-scaled, inversely quantized spectra

The output from the scalefactors tool is:

- The scaled, inversely quantized spectra

The M/S tool converts spectra pairs from Mid/Side to Left/Right under control of the M/S decision information, improving stereo imaging quality and sometimes providing coding efficiency.

The inputs to the M/S tool are:

- The M/S decision information
- The scaled, inversely quantized spectra related to pairs of channels

The output from the M/S tool is:

- The scaled, inversely quantized spectra related to pairs of channels, after M/S decoding

Note: The scaled, inversely quantized spectra of individually coded channels are not processed by the M/S block, rather they are passed directly through the block without modification. If the M/S block is not active, all spectra are passed through this block unmodified.

The prediction tool reverses the prediction process carried out at the encoder. This prediction process re-inserts the redundancy that was extracted by the prediction tool at the encoder, under the control of the predictor state information. This tool is implemented as a second order backward adaptive predictor.

The inputs to the prediction tool are:

- The predictor state information
- The predictor side information
- The scaled, inversely quantized spectra

The output from the prediction tool is:

- The scaled, inversely quantized spectra, after prediction is applied.

Note: If the prediction is disabled, the scaled, inversely quantized spectra are passed directly through the block without modification.

Alternatively, there is a forward adaptive long term prediction tool provided.

The inputs to the long term prediction tool are:

- The reconstructed time domain output of the decoder
- The scaled, inversely quantized spectra

The output from the long term prediction tool is:

- The scaled, inversely quantized spectra, after prediction is applied.

Note: If the prediction is disabled, the scaled, inversely quantized spectra are passed directly through the block without modification.

The perceptual noise substitution (PNS) tool implements noise substitution decoding on channel spectra by providing an efficient representation for noise-like signal components.

The inputs to the perceptual noise substitution tool are:

- The inversely quantized spectra
- The perceptual noise substitution control information

The output from the perceptual noise substitution tool is:

- The inversely quantized spectra

Note: If either part of this block is disabled, the scaled, inversely quantized spectra are passed directly through this part without modification. If the perceptual noise substitution block is not active, all spectra are passed through this block unmodified.

The intensity stereo tool implements intensity stereo decoding on pairs of spectra.

The inputs to the intensity stereo tool are:

- The inversely quantized spectra
- The intensity stereo control information

The output from the intensity stereo tool is:

- The inversely quantized spectra after intensity channel decoding.

Note:The scaled, inversely quantized spectra of individually coded channles are passed directly through this tool without modification. The intensity stereo tool and M/S tool are arranged so that the operation of M/S and intensity stereo are mutually exclusive on any given scalefactor band and group of one pair of spectra.

The coupling tool for dependently switched coupling channels adds the relevant data from dependently switched coupling channels to the spectra, as directed by the coupling control information.

The inputs to the coupling tool are:

- The inversely quantized spectra
- The coupling control information

The output from the coupling tool is:

- The inversely quantized spectra coupled with the dependently switched coupling channels.

Note: The scaled, inversely quantized spectra are passed directly through this tool without modification, if coupling is not indicated. Depending on the coupling control information, dependently switched coupling channels might either be coupled before or after the TNS processing.

The coupling tool for independently switched coupling channels adds the relevant data from independently switched coupling channels to the time signal, as directed by the coupling control information.

The inputs to the coupling tool are:

- The time signal as output by the filterbank
- The coupling control information

The output from the coupling tool is:

- The time signal coupled with the independently switched coupling channels.

Note: The time signal is passed directly through this tool without modification, if coupling is not indicated.

The temporal noise shaping (TNS) tool implements a control of the fine time structure of the coding noise. In the encoder, the TNS process has flattened the temporal envelope of the signal to which it has been applied. In the decoder, the inverse process is used to restore the actual temporal envelope(s), under control of the TNS information. This is done by applying a filtering process to parts of the spectral data.

The inputs to the TNS tool are:

- The inversely quantized spectra
- The TNS information

The output from the TNS block is:

- The inversely quantized spectra

Note: If this block is disabled, the inversely quantized spectra are passed through without modification.

The filterbank / block switching tool applies the inverse of the frequency mapping that was carried out in the encoder. An inverse modified discrete cosine transform (IMDCT) is used for the filterbank tool. The IMDCT can be configured to support either one set of 120, 128, 480, 512, 960, or 1024, or four sets of 32 or 256 spectral coefficients.

The inputs to the filterbank tool are:

- The inversely quantized spectra
- The filterbank control information

The output(s) from the filterbank tool is (are):

- The time domain reconstructed audio signal(s).

When present, the gain control tool applies a separate time domain gain control to each of 4 frequency bands that have been created by the gain control PQF filterbank in the encoder. Then, it assembles the 4 frequency bands and reconstructs the time waveform through the gain control tool's filterbank.

The inputs to the gain control tool are:

- The time domain reconstructed audio signal(s)
- The gain control information

The output(s) from the gain control tool is (are):

- The time domain reconstructed audio signal(s)

If the gain control tool is not active, the time domain reconstructed audio signal(s) are passed directly from the filterbank tool to the output of the decoder. This tool is used for the scalable sampling rate (SSR) audio object type only.

The SBR tool regenerates the highband of the audio signal. It is based on replication of the sequences of harmonics, truncated during encoding. It adjusts the spectral envelope of the generated high-band and applies inverse filtering, and adds noise and sinusoidal components in order to recreate the spectral characteristics of the original signal.

The input to the SBR tool is:

- The quantized envelope data;
- Misc. control data;
- A time domain signal from the AAC core decoder.

The output of the SBR tool is:

- A time domain signal.

The spectrum normalization tool converts the reconstructed flat spectra to the actual values at the decoder. The spectral envelope is specified by LPC coefficients, a Bark scale envelope, periodic peak components, and gain.

The input to the spectral normalization tool are

- The reconstructed flat spectra
- The information of LPC coefficients, a Bark scale envelope, periodic peak components and gain

The output from the spectral normalization tool is

- The reconstructed actual spectra

The interleaved VQ tool converts the vector index to the flattened spectra at the TwinVQ decoder by means of table look-up of the codebook and inverse interleaving of the spectra. Quantization noise is minimized by a weighted distortion measure at the encoder instead of an adaptive bit allocation. This is an alternative to the AAC quantization tool.

The input to the interleaved VQ tool is:

- A set of indices of the code vector.

The output from the TwinVQ tool is:

- The reconstructed flattened spectra

The Frequency Selective Switch (FSS) tool is used to control the combination of the AAC coding layer with both, TwinVQ, and CELP coding layer, if these are used as base layer coder in scalable configurations. In a second function this tool is applied to control the combination of mono and stereo coding layer in scalable configurations where both mono, and stereo coding layer are used to code a stereo input signal.

The Up-sampling Filter tool adapts the sampling rate of a CELP core coder, which can be used as base layer coder in scalable configurations, to the sampling rate of the AAC extension layer.

The input to the Upsampling Filter tool is:

- The output of a CELP core coder running at a lower sampling rate than the AAC extension layer

The output from the Up-sampling Filter tool is:

- The up-sampled CELP core coder output, matching the sampling rate of the AAC extension layer, transformed into the frequency domain with exactly the same frequency and time resolution as the AAC extension layer.

The BSAC noiseless decoding tool takes information from the bitstream payload demultiplexer, parses that information, decodes the Arithmetic coded data, and reconstructs the quantized spectra and the Arithmetic coded scalefactors. The BSAC noiseless coding module is an alternative to the AAC coding module. The BSAC noiseless coding is used to make the bitstream payload scalable and error resilient and further reduce the redundancy of the scalefactors and the quantized spectrum.

The inputs to the BSAC decoding tool are

- The Arithmetic model information for the noiselessly coded spectra
- The noiselessly coded bit-sliced data

The outputs from the BSAC decoding tool are

- The decoded integer representation of the scalefactors
- The quantized value for the spectra

The virtual codebooks (VCB11) tool can extend the part of the bitstream payload demultiplexer that decodes the sectioning information. The VCB11 tool gives the opportunity to detect serious errors within the spectral data of an MPEG-4 AAC bitstream payload.

The input of the VCB11 tool is:

- The encoded section data using virtual codebooks

The output of the VCB11 tool is:

- The decoded sectioning information

The reversible variable length coding (RVLC) tool can replace the part of the noiseless coding tool that decodes the Huffman and DPCM coded scalefactors. The RVLC tool is used to increase the error resilience for the scalefactor data within an MPEG-4 AAC bitstream payload.

The input of the RVLC tool is:

- The noiselessly coded scalefactors using RVLC

The output of the RVLC tool is:

- The decoded integer representation of the scalefactors

The Huffman codeword reordering (HCR) tool can extend the part of the noiseless coding tool that decodes the Huffman coded spectral data. The HCR tool is used to increase the error resilience for the spectral data within an MPEG-4 AAC bitstream payload.

The input of the HCR tool is:

- The sectioning information for the noiselessly coded spectra
- The noiselessly coded spectral data in an error resilient reordered manner
- The length of the longest codeword within spectral_data
- The length of spectral_data

The output of the HCR tool is:

- The quantized value of the spectra

4.2 Normative references

ISO/IEC 11172-3:1993, *Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s, Part 3: Audio*.

ITU-T Rec.H.222.0(1995) | ISO/IEC 13818-1:2000, *Information technology - Generic coding of moving pictures and associated audio information: – Part 1: Systems.*

ISO/IEC 13818-3:1997, *Information technology - Generic coding of moving pictures and associated audio information: - Part 3: Audio*.

ISO/IEC 13818-7, *Information technology - Generic coding of moving pictures and associated audio information: - Part 7: Advanced Audio Coding (AAC).*

4.3 Definitions

Definitions can be found in subpart 1, subclause 1.3.

4.4 Syntax

4.4.1 Decoder configuration (GASpecificConfig)

Table 4.1 – Syntax of GASpecificConfig()

Syntax	No. of bits	Mnemonic
GASpecificConfig (samplingFrequencyIndex, channelConfiguration, audioObjectType)		
{		
frameLengthFlag;	**1**	**bslbf**
dependsOnCoreCoder;	**1**	**bslbf**
if (dependsOnCoreCoder) {		
coreCoderDelay;	**14**	**uimsbf**
}		
extensionFlag;	**1**	**bslbf**
if (! channelConfiguration) {		

```
        program_config_element ();
    }
    if ((audioObjectType == 6) || (audioObjectType == 20)) {
        layerNr;                                                    3       uimsbf
    }
    if (extensionFlag) {
        if (audioObjectType == 22) {
            numOfSubFrame;                                          5       bslbf
            layer_length;                                           11      bslbf
        }
        if (audioObjectType == 17 || audioObjectType == 19 ||
            audioObjectType == 20 || audioObjectType == 23) {
            aacSectionDataResilienceFlag;                           1       bslbf
            aacScalefactorDataResilienceFlag;                       1       bslbf
            aacSpectralDataResilienceFlag;                          1       bslbf
        }
        extensionFlag3;                                             1       bslbf
        if (extensionFlag3) {
            /* tbd in version 3 */
        }
    }
}
```

4.4.1.1 Program config element

Table 4.2 – Syntax of program_config_element()

Syntax	No. of bits	Mnemonic
program_config_element()		
{		
element_instance_tag;	**4**	**uimsbf**
object_type;	**2**	**uimsbf**
sampling_frequency_index;	**4**	**uimsbf**
num_front_channel_elements;	**4**	**uimsbf**
num_side_channel_elements;	**4**	**uimsbf**
num_back_channel_elements;	**4**	**uimsbf**
num_lfe_channel_elements;	**2**	**uimsbf**
num_assoc_data_elements;	**3**	**uimsbf**
num_valid_cc_elements;	**4**	**uimsbf**
mono_mixdown_present;	**1**	**uimsbf**
if (mono_mixdown_present == 1)		
mono_mixdown_element_number;	**4**	**uimsbf**
stereo_mixdown_present;	**1**	**uimsbf**
if (stereo_mixdown_present == 1)		
stereo_mixdown_element_number;	**4**	**uimsbf**
matrix_mixdown_idx_present;	**1**	**uimsbf**
if (matrix_mixdown_idx_present == 1) {		
matrix_mixdown_idx ;	**2**	**uimsbf**
pseudo_surround_enable;	**1**	**uimsbf**
}		
for (i = 0; i < num_front_channel_elements; i++) {		
front_element_is_cpe[i];	**1**	**bslbf**
front_element_tag_select[i];	**4**	**uimsbf**
}		
for (i = 0; i < num_side_channel_elements; i++) {		

side_element_is_cpe[i];	**1**	**bslbf**
side_element_tag_select[i];	**4**	**uimsbf**
}		
for (i = 0; i < num_back_channel_elements; i++) {		
back_element_is_cpe[i];	**1**	**bslbf**
back_element_tag_select[i];	**4**	**uimsbf**
}		
for (i = 0; i < num_lfe_channel_elements; i++)		
lfe_element_tag_select[i];	**4**	**uimsbf**
for (i = 0; i < num_assoc_data_elements; i++)		
assoc_data_element_tag_select[i];	**4**	**uimsbf**
for (i = 0; i < num_valid_cc_elements; i++) {		
cc_element_is_ind_sw[i];	**1**	**uimsbf**
valid_cc_element_tag_select[i];	**4**	**uimsbf**
}		
byte_alignment(); [a]		
comment_field_bytes;	**8**	**uimsbf**
for (i = 0; i < comment_field_bytes; i++)		
comment_field_data[i];	**8**	**uimsbf**
}		

[a] If called from within an AudioSpecificConfig(), this byte_alignment shall be relative to the start of the AudioSpecificConfig().

4.4.2 GA bitstream payloads

4.4.2.1 Payloads for the audio object types AAC main, AAC SSR, AAC LC and AAC LTP

Table 4.3 – Syntax of top level payload for audio object types AAC Main, SSR, LC, and LTP (raw_data_block())

Syntax	No. of bits	Mnemonic
raw_data_block()		
{		
while((id = **id_syn_ele**) != ID_END){	**3**	**uimsbf**
switch (id) {		
case ID_SCE: single_channel_element();		
break;		
case ID_CPE: channel_pair_element();		
break;		
case ID_CCE: coupling_channel_element();		
break;		
case ID_LFE: lfe_channel_element();		
break;		
case ID_DSE: data_stream_element();		
break;		
case ID_PCE: program_config_element();		
break;		
case ID_FIL: fill_element();		
}		
}		
byte_alignment();		
}		

Table 4.4 – Syntax of single_channel_element()

Syntax	No. of bits	Mnemonic
single_channel_element()		
{		
element_instance_tag;	**4**	**uimsbf**

```
    individual_channel_stream(0,0);
}
```

Table 4.5 – Syntax of channel_pair_element()

Syntax	No. of bits	Mnemonic
channel_pair_element()		
{		
element_instance_tag;	**4**	**uimsbf**
common_window;	**1**	**uimsbf**
if (common_window) {		
ics_info();		
ms_mask_present;	**2**	**uimsbf**
if (ms_mask_present == 1) {		
for (g = 0; g < num_window_groups; g++) {		
for (sfb = 0; sfb < max_sfb; sfb++) {		
ms_used[g][sfb**];**	**1**	**uimsbf**
}		
}		
}		
}		
individual_channel_stream(common_window,0);		
individual_channel_stream(common_window,0);		
}		

Table 4.6 – Syntax of ics_info()

Syntax	No. of bits	Mnemonic
ics_info()		
{		
ics_reserved_bit;	**1**	**bslbf**
window_sequence;	**2**	**uimsbf**
window_shape;	**1**	**uimsbf**
if (window_sequence == EIGHT_SHORT_SEQUENCE) {		
max_sfb;	**4**	**uimsbf**
scale_factor_grouping;	**7**	**uimsbf**
}		
else {		
max_sfb;	**6**	**uimsbf**
predictor_data_present;	**1**	**uimsbf**
if (predictor_data_present) {		
if (audioObjectType == 1) {		
predictor_reset;	**1**	**uimsbf**
if (predictor_reset) {		
predictor_reset_group_number;	**5**	**uimsbf**
}		
for (sfb = 0; sfb < min (max_sfb, PRED_SFB_MAX); sfb++) {		
prediction_used[sfb**];**	**1**	**uimsbf**
}		
}		
else {		
ltp_data_present;	**1**	**uimsbf**
if (ltp_data_present) {		
ltp_data();		

```
                    }
                    if (common_window) {
                        ltp_data_present;                    1          uimsbf
                        if (ltp_data_present) {
                            ltp_data();
                        }
                    }
                }
            }
        }
    }
```

Table 4.7 – Syntax of pulse_data()

Syntax	No. of bits	Mnemonic
pulse_data() {		
number_pulse;	**2**	**uimsbf**
pulse_start_sfb;	**6**	**uimsbf**
for (i = 0; i < number_pulse+1; i++) {		
pulse_offset[i];	**5**	**uimsbf**
pulse_amp[i];	**4**	**uimsbf**
}		
}		

Table 4.8 – Syntax of coupling_channel_element()

Syntax	No. of bits	Mnemonic
coupling_channel_element()		
{		
element_instance_tag;	**4**	**uimsbf**
ind_sw_cce_flag;	**1**	**uimsbf**
num_coupled_elements;	**3**	**uimsbf**
num_gain_element_lists = 0;		
for (c = 0; c < num_coupled_elements+1; c++) {		
num_gain_element_lists++;		
cc_target_is_cpe[c];	**1**	**uimsbf**
cc_target_tag_select[c];	**4**	**uimsbf**
if (cc_target_is_cpe[c]) {		
cc_l[c];	**1**	**uimsbf**
cc_r[c];	**1**	**uimsbf**
if (cc_l[c] && cc_r[c])		
num_gain_element_lists++;		
}		
}		
cc_domain;	**1**	**uimsbf**
gain_element_sign;	**1**	**uimsbf**
gain_element_scale;	**2**	**uimsbf**
individual_channel_stream(0,0);		
for (c=1; c<num_gain_element_lists; c++) {		
if (ind_sw_cce_flag) {		
cge = 1;		
} else {		
common_gain_element_present[c];	**1**	**uimsbf**
cge = common_gain_element_present[c];		

```
        }
        if (cge)
            hcod_sf[common_gain_element[c]];                          1..19    vlclbf
        else {
            for (g = 0; g < num_window_groups; g++) {
                for (sfb=0; sfb<max_sfb; sfb++) {
                    if (sfb_cb[g][sfb] != ZERO_HCB)
                        hcod_sf[dpcm_gain_element[c][g][sfb]];        1..19    vlclbf
                }
            }
        }
    }
}
```

Table 4.9 – Syntax of lfe_channel_element()

Syntax	No. of bits	Mnemonic
lfe_channel_element()		
{		
element_instance_tag;	**4**	**uimsbf**
individual_channel_stream(0,0);		
}		

Table 4.10 – Syntax of data_stream_element()

Syntax	No. of bits	Mnemonic
data_stream_element()		
{		
element_instance_tag;	**4**	**uimsbf**
data_byte_align_flag;	**1**	**uimsbf**
cnt = **count;**	**8**	**uimsbf**
if (cnt == 255)		
cnt += **esc_count**;	**8**	**uimsbf**
if (data_byte_align_flag)		
byte_alignment();		
for (i = 0; i < cnt; i++)		
data_stream_byte[element_instance_tag][i];	**8**	**uimsbf**
}		

Table 4.11 – Syntax of fill_element()

Syntax	No. of bits	Mnemonic
Fill_element()		
{		
cnt = **count;**	**4**	**uimsbf**
if (cnt == 15)		
cnt += **esc_count** - 1;	**8**	**uimsbf**
while (cnt > 0) {		
cnt -= extension_payload(cnt);		
}		
}		

Table 4.12 – Syntax of gain_control_data()

Syntax	No. of bits	Mnemonic
gain_control_data()		
{		
max_band;	**2**	**uimsbf**

```
    if (window_sequence == ONLY_LONG_SEQUENCE) {
        for (bd = 1; bd <= max_band; bd++) {
            for (wd = 0; wd < 1; wd++) {
                adjust_num[bd][wd];                                  3        uimsbf
                for (ad = 0; ad < adjust_num[bd][wd]; ad++) {
                    alevcode[bd][wd][ad];                            4        uimsbf
                    aloccode[bd][wd][ad];                            5        uimsbf
                }
            }
        }
    }
    else if (window_sequence == LONG_START_SEQUENCE) {
        for (bd = 1; bd <= max_band; bd++) {
            for (wd = 0; wd < 2; wd++) {
                adjust_num[bd][wd];                                  3        uimsbf
                for (ad = 0; ad < adjust_num[bd][wd]; ad++) {
                    alevcode[bd][wd][ad];                            4        uimsbf
                    if (wd == 0)
                        aloccode[bd][wd][ad];                        4        uimsbf
                    else
                        aloccode[bd][wd][ad];                        2        uimsbf
                }
            }
        }
    }
    else if (window_sequence == EIGHT_SHORT_SEQUENCE) {
        for (bd = 1; bd <= max_band; bd++) {
            for(wd = 0; wd < 8; wd++) {
                adjust_num[bd][wd];                                  3        uimsbf
                for (ad = 0; ad < adjust_num[bd][wd]; ad++) {
                    alevcode[bd][wd][ad];                            4        uimsbf
                    aloccode[bd][wd][ad];                            2        uimsbf
                }
            }
        }
    }
    else if (window_sequence == LONG_STOP_SEQUENCE) {
        for (bd = 1; bd <= max_band; bd++) {
            for (wd = 0; wd < 2; wd++) {
                adjust_num[bd][wd];                                  3        uimsbf
                for (ad = 0; ad < adjust_num[bd][wd]; ad++) {
                    alevcode[bd][wd][ad];                            4        uimsbf
                    if (wd == 0)
                        aloccode[bd][wd][ad];                        4        uimsbf
                    else
                        aloccode[bd][wd][ad];                        5        uimsbf
                }
            }
        }
    }
}
```

4.4.2.2 Payloads for the audio object type AAC scalable

Table 4.13 – Syntax of the ASME top level payload for the audio object type AAC scalable (aac_scalable_main_element)

Syntax	No. of bits	Mnemonic

aac_scalable_main_element()
{
aac_scalable_main_header();
for (ch=0; ch<(this_layer_stereo ? 2:1); ch++){
individual_channel_stream(1, 1);
}
cnt = bits_to_decode() / 8;
while (cnt >= 1) {
cnt -= extension_payload(cnt);
}
byte_alignment();
}		

Table 4.14 – Syntax of the ASEE top level payload for the audio object type AAC scalable (aac_scalable_extension_element)

Syntax	No. of bits	Mnemonic

aac_scalable_extension_element()
{
aac_scalable_extension_header()
for (ch=0; ch<(this_layer_stereo ? 2:1); ch++){
individual_channel_stream(1, 1)
}
cnt = bits_to_decode() / 8
while (cnt >= 1) {
cnt -= extension_payload(cnt)
}
byte_alignment();
}		

Table 4.15 – Syntax of aac_scalable_main_header()

Syntax	No. of bits	Mnemonic
aac_scalable_main_header() {		
ics_reserved_bit;	**1**	**bslbf**
If (tvq_layer_present == 0) {		
window_sequence;	**2**	**uimsbf**
window_shape;	**1**	**uimsbf**
}		
if (window_sequence == EIGHT_SHORT_SEQUENCE) {		
max_sfb;	**4**	**uimsbf**
scale_factor_grouping;	**7**	**uimsbf**
} else {		
max_sfb;	**6**	**uimsbf**
}		
if (this_layer_stereo) {		
ms_mask_present;	**2**	**bslbf**
if (ms_mask_present == 1) {		
ms_data();		
}		
}		
if (mono_stereo_flag && (core_flag		

```
            || (tvq_layer_present && tvq_mono_tns == 0)))
            tns_channel_mono_layer;                          1          bslbf

      for (ch = 0; ch < (this_layer_stereo ? 2:1 ); ch++) {
            if (!tvq_layer_present  || (tns_aac_tvq_en[ch] == 1)) {
                  tns_data_present;                          1          bslbf
                  if (tns_data_present)
                        tns_data();
            }
            if (core_flag || tvq_layer_present) {
                  if ((ch == 0) || ((ch == 1) && (core_stereo ||
                        diff_control_data();
                  if (mono_stereo_flag)
                        diff_control_data_lr();
            } else {
                  ltp_data_present;                          1          bsblf
                  if (ltp_data_present) {
                        ltp_data ();
                  }
            }
      }
}
```

Table 4.16 – Syntax of aac_scalable_extension_header()

Syntax	No. of bits	Mnemonic
aac_scalable_extension_header()		
{		
if (window_sequence == EIGHT_SHORT_SEQUENCE) {		
max_sfb;	**4**	**bslbf**
} else {		
max_sfb;	**6**	**bslbf**
}		
if (this_layer_stereo) {		
ms_mask_present;	**2**	**bslbf**
If (ms_mask_present == 1) {		
ms_data();		
}		
}		
if (mono_stereo_flag) {		
for (ch = 0; ch < 2; ch++) {		
tns_data_present;	**1**	**bslbf**
if (tns_data_present)		
tns_data();		
}		
}		
if ((mono_layer_flag) && (this_layer_stereo)) {		
for (ch = 0; ch < 2; ch++) {		
diff_control_data_lr();		
}		
}		
}		

Table 4.17 – Syntax of diff_control_data()

Syntax	No. of bits	Mnemonic
diff_control_data()		

{			
If (window_sequence == EIGHT_SHORT_SEQUENCE)			
for (win = 0; win < 8; w++)			
diff_control[win][0];		**1**	**bslbf**
else			
for (dc_group=0; dc_group<no_of_dc_groups;			
dc_group++);			
diff_control[0][dc_group];		**2..5**	**bslbf**
}			

Table 4.18 – Syntax of diff_control_data_lr()

Syntax	No. of bits	Mnemonic
diff_control_data_lr()		
{		
if (window_sequence != EIGHT_SHORT_SEQUENCE) {		
for (sfb = last_max_sfb_ms;		
sfb < min(last_mono_max_sfb;max_sfb); sfb++)		
if (!ms_used[0][sfb])		
diff_control_lr[0][sfb];	**1**	**bslbf**
} else {		
if (last_max_sfb_ms == 0) /* only in the first stereo layer*/		
for (win = 0; win < 8; win++)		
diff_control_lr[win][0];	**1**	**bslbf**
}		
}		

4.4.2.3 Payloads for the audio object types ER AAC LC, ER AAC LTP and ER AAC LD

Table 4.19 – Syntax of top level payload for audio object types ER AAC LC, ER AAC LTP and ER AAC LD (er_raw_data_block())

Syntax	No. of bits	Mnemonic
er_raw_data_block()		
{		
if (channelConfiguration == 0) {		
/* reserved */		
}		
if (channelConfiguration == 1) {		
single_channel_element();		
}		
if (channelConfiguration == 2) {		
channel_pair_element();		
}		
if (channelConfiguration == 3) {		
single_channel_element();		
channel_pair_element();		
}		
if (channelConfiguration == 4) {		
single_channel_element();		
channel_pair_element();		
single_channel_element();		
}		
if (channelConfiguration == 5) {		

```
            single_channel_element();
            channel_pair_element();
            channel_pair_element();
        }
        if (   channelConfiguration == 6 ) {
            single_channel_element();
            channel_pair_element();
            channel_pair_element();
            lfe_channel_element();
        }
        if (   channelConfiguration == 7 ) {
            single_channel_element();
            channel_pair_element();
            channel_pair_element();
            channel_pair_element();
            lfe_channel_element();
        }
        if (   channelConfiguration >= 8 ) {
            /* reserved */
        }
        cnt = bits_to_decode() / 8;
        while ( cnt >= 1 ) {
            cnt -= extension_payload(cnt);
        }
        byte_alignment();
}
```

4.4.2.4 Payloads for the audio object type Twin_VQ

Table 4.20 – Syntax of the TSME top level payload for the audio object type Twin_VQ (tvq_scalable_main_element)

Syntax	No. of bits	Mnemonic

tvq_scalable_main_element()
{
  tvq_scalable_main_header();
  vq_single_element(0);
}		

Table 4.21 – Syntax of the TSEE top level payload for the audio object type Twin_VQ (tvq_scalable_extension_element)

Syntax	No. of bits	Mnemonic

tvq_scalable_extension_element()
{
  tvq_scalable_extension_header();
  vq_single_element(lay);
}		

Table 4.22 – Syntax of tvq_scalable_main_header()

Syntax	No. of bits	Mnemonic
tvq_scalable_main_header() {		
window_sequence;	**2**	**bslbf**
window_shape;	**1**	**bslbf**

```
    if (this_layer_stereo) {
        ms_mask_present;                                  2        bslbf
        if (ms_mask_present == 1) {
            if (window_sequence ==
                EIGHT_SHORT_SEQUENCE)
                scale_factor_grouping;                    7        bslbf
            ms_data();
        }
    }

    for( ch = 0; ch < (this_layer_stereo ? 2:1); ch++ ) {
        ltp_data_present;                                 1        bslbf
        if (ltp_data_present)
            ltp_data ();

        tns_data_present;                                 1        bslbf
        if (tns_data_present)
            tns_data();
    }
}
```

Table 4.23 – Syntax of tvq_scalable_extension_header()

Syntax	No. of bits	Mnemonic
tvq_scalable_extension_header()		
{		
if (this_layer_stereo) {		
ms_mask_present;	**2**	**bslbf**
if (ms_mask_present == 1) {		
ms_data();		
}		
}		
}		

Table 4.24 – Syntax of vq_single_element

Syntax	No. of bits	Mnemonic
vq_single_element(lyr)		
{		
if (lyr == 0);		
bandlimit_present	**1**	**uimsbf**
if (window_sequence != EIGHT_SHORT_SEQUENCE		
&& lyr == 0) {		
ppc_present;	**1**	**uimsbf**
postprocess _present;	**1**	**uimsbf**
}		
if (lyr >= 1)		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		
fb_shift[i_ch];	**2**	**uimsbf**
}		
if (lyr == 0 && **bandlimit_present**) {		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		
index_blim_h[i_ch];	**2**	**uimsbf**
index_blim_l[i_ch];	**1**	**uimsbf**
}		
}		
if (**ppc_present**) {		

```
        for (idiv = 0; idiv < N_DIV_P; idiv++) {
            index_shape0_p[idiv];                          7          uimsbf
            index_shape1_p[idiv];                          7          uimsbf
        }
        for (i_ch = 0; i_ch < n_ch; i_ch++) {
            index_pit[i_ch];                               8          uimsbf
            index_pgain[i_ch];                             7          uimsbf
        }
    }
    for (idiv = 0; idiv < N_DIV; idiv++) {
        index_shape0[idiv];                                5/6        uimsbf
        index_shape1[idiv];                                5/6        uimsbf
    }
    for (i_ch = 0; i_ch < n_ch; i_ch++) {
        for (isb = 0; isb < N_SF; isb++) {
            for (ifdiv = 0; ifdiv < FW_N_DIV; ifdiv++) {
                index_env[i_ch][isb][ifdiv];               0,6        uimsbf
            }
        }
    }
    for (i_ch = 0; i_ch < n_ch; i_ch++) {
        for (isbm = 0; isbm < N_SF; isbm++){
            index_fw_alf[i_ch][isbm];                      0,1        uimsbf
        }
    }
    for (i_ch = 0; i_ch < n_ch; i_ch++){
        index_gain[i_ch]                                   8..9       uimsbf
        if (N_SF[b_type] > 1){
            for (isbm = 0; isbm < N_SF[b_type]; isbm++) {
                index_gain_sb[i_ch][isbm]                  4          uimsbf
            }
        }
    }
    for (i_ch = 0; i_ch < n_ch; i_ch++) {
        index_lsp0[i_ch]                                   1          uimsbf
        index_lsp1[i_ch]                                   6          uimsbf
        for (isplt = 0; isplt < LSP_SPLIT; isplit++) {
            index_lsp2[i_ch][isplt]                        4          uimsbf
        }
    }
}
```

4.4.2.5 Payloads for the audio object type ER TwinVQ

Table 4.25 – Syntax of ER TwinVQ object type (base)

Syntax	No. of bits	Mnemonic

tvq_scalable_main_element()
{
  Error_Sensitivity_Category1();
  Error_Sensitivity_Category2();
}		

Table 4.26 – Syntax of ER TwinVQ object type (enhancement)

Syntax	No. of bits	Mnemonic
tvq_scalable_extension_element()		

```
{
    Error_Sensitivity_Category3();
    Error_Sensitivity_Category4();
}
```

Table 4.27 – Syntax of Error_Sensitivity_Category1()

Syntax	No. of bits	Mnemonic
Error_Sensitivity_Category1()		
{		
window_sequence;	**2**	**bslbf**
window_shape;	**1**	**bslbf**
if (this_layer_stereo) {		
ms_mask_present;	**2**	**bslbf**
if (ms_mask_present == 1) {		
if (window_sequence ==		
EIGHT_SHORT_SEQUENCE)		
scale_factor_grouping;	**7**	**bslbf**
ms_data();		
}		
}		
for (ch = 0; ch < (this_layer_stereo ? 2:1); ch++) {		
ltp_data_present;	**1**	**bsblf**
if (ltp_data_present**)**		
ltp_data ();		
tns_data_present;	**1**	**bslbf**
if (tns_data_present)		
tns_data();		
}		
bandlimit_present;	**1**	**uimsbf**
if (window_sequence != EIGHT_SHORT_SEQUENCE) {		
ppc_present;	**1**	**uimsbf**
postprocess_present;	**1**	**uimsbf**
}		
if (bandlimit_present) {		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		
index_blim_h[i_ch];	**2**	**uimsbf**
index_blim_l[i_ch];	**1**	**uimsbf**
}		
}		
if (ppc_present) {		
for (idiv = 0; idiv < N_DIV_P; idiv++) {		
index_shape0_p[idiv];	**7**	**uimsbf**
index_shape1_p[idiv];	**7**	**uimsbf**
}		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		
index_pit[i_ch];	**8**	**uimsbf**
index_pgain[i_ch];	**7**	**uimsbf**
}		
}		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		

	index_gain[i_ch];	**9**	**uimsbf**
	if (N_SF[b_type] > 1) {		
	for (isbm = 0; isbm < N_SF[b_type]; isbm++) {		
	index_gain_sb[i_ch][isbm];	**4**	**uimsbf**
	}		
	}		
}			
	for (i_ch = 0; i_ch < n_ch; i_ch++) {		
	index_lsp0[i_ch];	**1**	**uimsbf**
	index_lsp1[i_ch];	**6**	**uimsbf**
	for (isplt = 0; isplt < LSP_SPLIT; isplit++) {		
	index_lsp2[i_ch][isplt];	**4**	**uimsbf**
	}		
}			
	for (i_ch = 0; i_ch < n_ch; i_ch++) {		
	for (isb = 0; isb < N_SF; isb++) {		
	for (ifdiv = 0; ifdiv < FW_N_DIV; ifdiv++) {		
	index_env[i_ch][isb][ifdiv];	**0,6**	**uimsbf**
	}		
	}		
}			
	for (i_ch = 0; i_ch < n_ch; i_ch++) {		
	for (isbm = 0; isbm < N_SF; isbm++) {		
	index_fw_alf[i_ch][isbm];	**0,1**	**uimsbf**
	}		
}			
}			

Table 4.28 – Syntax of Error_Sensitivity_Category2()

Syntax	No. of bits	Mnemonic
Error_Sensitivity_Category2()		
{		
for (idiv = 0; idiv < N_DIV; idiv++) {		
index_shape0[idiv];	**5/6**	**uimsbf**
index_shape1[idiv];	**5/6**	**uimsbf**
}		
}		

Table 4.29 – Syntax of Error_Sensitivity_Category3()

Syntax	No. of bits	Mnemonic
Error_Sensitivity_Category3()		
{		
if (this_layer_stereo) {		
ms_mask_present;	**2**	**bslbf**
if (ms_mask_present == 1) {		
ms_data();		
}		
}		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		
fb_shift[i_ch];	**2**	**uimsbf**

}		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		
index_gain[i_ch];	**8**	**uimsbf**
if (N_SF[b_type] > 1) {		
for (isbm = 0; isbm < N_SF[b_type]; isbm++) {		
index_gain_sb[i_ch][isbm];	**4**	**uimsbf**
}		
}		
}		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		
index_lsp0[i_ch];	**1**	**uimsbf**
index_lsp1[i_ch];	**6**	**uimsbf**
for (isplt = 0; isplt < LSP_SPLIT; isplit++) {		
index_lsp2[i_ch][isplt];	**4**	**uimsbf**
}		
}		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		
for (isb = 0; isb < N_SF; isb++) {		
for (ifdiv = 0; ifdiv < FW_N_DIV; ifdiv++) {		
index_env[i_ch][isb][ifdiv];	**0,6**	**uimsbf**
}		
}		
}		
for (i_ch = 0; i_ch < n_ch; i_ch++) {		
for (isbm = 0; isbm < N_SF; isbm++) {		
index_fw_alf[i_ch][isbm];	**0,1**	**uimsbf**
}		
}		
}		

Table 4.30 – Syntax of Error_Sensitivity_Category4()

Syntax	No. of bits	Mnemonic
Error_Sensitivity_Category4()		
{		
for (idiv = 0; idiv < N_DIV; idiv++) {		
index_shape0[idiv];	**5/6**	**uimsbf**
index_shape1[idiv];	**5/6**	**uimsbf**
}		
}		

4.4.2.6 Payloads for the audio object type ER BSAC

Table 4.31 – Syntax of top level payload for audio object type ER BSAC (bsac_payload())

Syntax	No. of bits	Mnemonic
bsac_payload(lay)		
{		
for (frm = 0; frm < numOfSubFrame; frm++) {		
bsac_lstep_element(frm, lay);		
}		

```
/*
bsac_lstep_element(frm, lay) should be mapped to the fine
grain audio data, bsac_raw_data_block(), for the actual
decoding. See subclause “Decoding of payload for audio object
type ER BSAC” for more detailed description.*/
}
```

Table 4.32 – Syntax of bsac_lstep_element()

Syntax	No. of bits	Mnemonic

bsac_lstep_element(frm, lay)
{
offset = LayerStartByte[frm][lay];
for(i = 0;i < LayerLength[frm][lay]; i++)
bsac_stream_byte[frm][offset+i];
	8	**uimsbf**

/* bsac_stream_byte should be mapped to the fine grain audio data, bsac_raw_data_block(), for the actual decoding. See subclause “Decoding of payload for audio object type ER BSAC” for more detailed description. */ }		

Table 4.33 – bsac_raw_data_block()

Syntax	No. of bits	Mnemonic

bsac_raw_data_block()
{
bsac_base_element();
layer = slayer_size;
while (data_available() && layer < (top_layer+slayer_size)) {
bsac_layer_element(nch, layer);
layer++;
}
byte_alignment();
}		

Table 4.34 – Syntax of bsac_base_element()

Syntax	No. of bits	Mnemonic
bsac_base_element() { **frame_length;**	**11**	**uimbf**

bsac_header();
general_header();
byte_alignment();
for (slayer = 0; slayer < slayer_size; slayer++)
bsac_layer_element(slayer);
}		

Table 4.35 – Syntax of bsac_header()

Syntax	No. of bits	Mnemonic
bsac_ header() { **header_length;**	**4**	**uimbf**

sba_mode;	**1**	**uimbf**
top_layer;	**6**	**uimbf**
base_snf_thr;	**2**	**uimbf**
for (ch = 0;ch < nch; ch++)		
max_scalefactor[ch];	**8**	**uimbf**
base_band;	**5**	**uimbf**
for(ch = 0;ch < nch; ch++) {		
cband_si_type[ch];	**5**	**uimbf**
base_scf_model[ch];	**3**	**uimbf**
enh_scf_model[ch];	**3**	**uimbf**
max_sfb_si_len[ch];	**4**	**uimbf**
}		
}		

Table 4.36 – Syntax of general_header()

Syntax	No. of bits	Mnemonic
general_header()		
{		
reserved_bit;	**1**	**bslbf**
window_sequence;	**2**	**uimsbf**
window_shape;	**1**	**uimsbf**
if (window_sequence == EIGHT_SHORT_SEQUENCE) {		
max_sfb;	**4**	**uimsbf**
scale_factor_grouping;	**7**	**uimsbf**
} else {		
max_sfb;	**6**	**uimsbf**
}		
pns_data_present;	**1**	**uimbf**
if (pns_data_present)		
pns_start_sfb;	**6**	**uimbf**
if (nch == 2)		
ms_mask_present;	**2**	**bslbf**
for (ch = 0 ch < nch; ch++) {		
tns_data_present[ch];	**1**	**bslbf**
if (tns_data_present[ch])		
tns_data();		
ltp_data_present[ch];	**1**	**bslbf**
if (ltp_data_present[ch])		
ltp_data(last_max_sfb, max_sfb);		
}		
}		

Table 4.37 – Syntax of bsac_layer_element()

Syntax	No. of bits	Mnemonic
bsac_layer_element(layer)		

```
{
    layer_cband_si(layer);
    layer_sfb_si(layer);

    bsac_layer_spectra (layer);
    if (!sba_mode) {
        bsac_lower_spectra (layer);
    }
    else if (terminal_layer[layer]) {
        bsac_lower_spectra (layer);
        bsac_higher_spectra (layer);
    }
}
```

Table 4.38 – Syntax of layer_cband_si()

Syntax	No. of bits	Mnemonic
layer_cband_si(layer)		
{		
g = layer_group[layer];		
for (ch = 0; ch < nch; ch++) {		
for (cband = layer_start_cband[layer];		
cband < layer_end_cband[layer]; cband++) {		
acode_cband_si[ch][g][cband];	**1..14**	**bslbf**
}		
}		
}		

Table 4.39 – Syntax of layer_sfb_si()

Syntax	No. of bits	Mnemonic
layer_sfb_si (layer)		
{		
g = layer_group[layer];		
for (ch = 0; ch < nch; ch++)		
for (sfb = layer_start_sfb[layer];sfb < layer_end_sfb[layer]; sfb++) {		
if (nch == 1) {		
if (pns_data_present && sfb >= pns_start_sfb) {		
acode_noise_flag[g][sfb];	**1**	**bslbf**
}		
} else if (stereo_side_info_coded[g][sfb] == 0) {		
if (ms_mask_present !=2) {		
if (ms_mask_present == 1) {		
acode_ms_used[g][sfb];	**0..2**	**bslbf**
} else if (ms_mask_present == 3) {		
acode_stereo_info[g][sfb];	**0..4**	**bslbf**
}		
if (pns_data_present && sfb >= pns_start_sfb) {		
acode_noise_flag_l[g][sfb];	**1**	**bslbf**
acode_noise_flag_r[g][sfb];	**1**	**bslbf**
if (ms_mask_present == 3 && stereo_info == 3) {		
if (noise_flag_l && noise_flag_r){		
acode_noise_mode[g][sfb];	**2**	**bslbf**
}		

```
                        }
                    }
                }
                stereo_side_info_coded[g][sfb] = 1;
            }
            if (noise_flag[ch][g][sfb]) {
                if (noise_pcm_flag[ch] == 1) {
                    acode_max_noise_energy[ch];                    9        bslbf
                    noise_pcm_flag[ch] = 0;
                }
                acode_dpcm_noise_energy_index[ch][g][sfb];         0..14    bslbf
            } else if (stereo_info[g][sfb] >=2 && ch == 1) {
                acode_is_position_index[g][sfb];                   0..14    bslbf
            } else {
                acode_scf_index[ch][g][sfb];                       0..14    slbf
            }
        }
}
```

Table 4.40 – Syntax of bsac_layer_spectra()

Syntax	No. of bits	Mnemonic

bsac_layer_spectra(layer)
{
g = layer_group[layer];
start_index[g] = layer_start_ index[layer];
end_index[g] = layer_end_ index[layer];
if (layer < slayer_size)
thr_snf = base_snf_thr;
else
thr_snf = 0;
bsac_spectral_data (g, g+1, thr_snf, cur_snf);
}		

Table 4.41 – Syntax of bsac_lower_spectra()

Syntax	No. of bits	Mnemonic

bsac_lower_spectra(layer)
{
for (g = 0; g < num_window_groups; g++) {
start_index[g] = 0;
end_index[g] = 0;
}
for (play = 0; play < layer; play++) {
end_index[layer_group[play]] = layer_end_index[play];
}
bsac_spectral_data (0, num_window_groups, 0, unc_snf);
}		

Table 4.42 – Syntax of bsac_higher_spectra()

Syntax	No. of bits	Mnemonic

bsac_higher_spectra(layer)
{
for (nlay = layer+1; nlay < top_layer+slayer_size; nlay++) {		

```
            g = layer_group[nlay];
            start_index[g] = layer_start_index[nlay];
            end_index[g]  = layer_end_index[nlay];
            bsac_spectral_data (g, g+1, 0, unc_snf);
        }
}
```

Table 4.43 – Syntax of bsac_spectral_data ()

Syntax	No. of bits	Mnemonic
bsac_spectral_data(start_g, end_g, thr_snf, cur_snf)		
{		
if (!layer_data_available()) return;		
for (snf = maxsnf; snf > thr_snf; snf--)		
for (g = start_g; g < end_g; g++)		
for (i = start_index[g]; i < end_index[g]; i++)		
for (ch = 0; ch < nch; ch++) {		
if (cur_snf[ch][g][i] < snf) continue;		
if (!sample[ch][g][i] || sign_is_coded[ch] [g][i])		
acod_sliced_bit[ch][g][i][snf];	**0..6**	**bslbf**
if (sample[ch][g][i] && !sign_is_coded[ch] [g][i]) {		
if (layer_data_available()) return;		
acod_sign[ch][g][i];	**1**	**bslbf**
sign_is_coded[ch][g][i] = 1;		
}		
cur_snf[ch][g][i]--;		
if (layer_data_available()) return;		
}		
}		

4.4.2.7 Subsidiary payloads

Table 4.44 – Syntax of individual_channel_stream()

Syntax	No. of bits	Mnemonic
individual_channel_stream(common_window, scale_flag)		
{		
global_gain;	**8**	**uimsbf**
if (! common_window && ! scale_flag) {		
ics_info ();		
}		
section_data ();		
scale_factor_data ();		
if (! scale_flag) {		
pulse_data_present;	**1**	**uimsbf**
if (pulse_data_present) {		
pulse_data ();		
}		
tns_data_present;	**1**	**uimsbf**
if (tns_data_present) {		
tns_data ();		
}		

	gain_control_data_present;	**1**	**uimsbf**
	if (gain_control_data_present) {		
	gain_control_data ();		
	}		
}			
if (! aacSpectralDataResilienceFlag) {			
	spectral_data ();		
}			
else {			
	length_of_reordered_spectral_data;	**14**	**uimsbf**
	length_of_longest_codeword;	**6**	**uimsbf**
	reordered_spectral_data ();		
}			
}			

Table 4.45 – Syntax of reordered_spectral_data ()

Syntax	No. of bits	Mnemonic
reordered_spectral_data ()		
{		
/* complex reordering, see tool description of Huffman codeword reordering (subclause 4.6.16.3) */		
}		

Table 4.46 – Syntax of section_data()

Syntax	No. of bits	Mnemonic
section_data()		
{		
if (window_sequence == EIGHT_SHORT_SEQUENCE) {		
sect_esc_val = (1 << 3) – 1;		
}		
else {		
sect_esc_val = (1 << 5) – 1;		
}		
for (g = 0; g < num_window_groups; g++) {		
k = 0;		
i = 0;		
while (k < max_sfb) {		
if (aacSectionDataResilienceFlag)		
sect_cb[g][i];	**5**	**uimsbf**
}		
else {		
sect_cb[g][i];	**4**	**uimsbf**
}		
sect_len = 0;		
if (! aacSectionDataResilienceFlag ||		
sect_cb < 11 || (sect_cb > 11 && sect_cb < 16)) {		
while (**sect_len_incr** == sect_esc_val) {	**3/5**	**uimsbf**
sect_len += sect_esc_val;		
}		
}		
else {		
sect_len_incr = 1;		
}		

```
                sect_len += sect_len_incr;
                sect_start[g][i] = k;
                sect_end[g][i] = k + sect_len;
                for (sfb = k; sfb < k + sect_len; sfb++) {
                    sfb_cb[g][sfb] = sect_cb[g][i];
                }
                k += sect_len;
                i++;
            }
            num_sec[g] = i;
        }
    }
```

Table 4.47 – Syntax of scale_factor_data()

Syntax	No. of bits	Mnemonic
scale_factor_data()		
{		
if (! aacScalefactorDataResilienceFlag) {		
noise_pcm_flag = 1;		
for (g = 0; g < num_window_groups; g++) {		
for (sfb = 0; sfb < max_sfb; sfb++) {		
if (sfb_cb[g][sfb] != ZERO_HCB) {		
if (is_intensity (g, sfb)) {		
hcod_sf[dpcm_is_position[g][sfb]];	**1..19**	**vlclbf**
}		
else {		
if (is_noise(g, sfb)) {		
if (noise_pcm_flag) {		
noise_pcm_flag = 0;		
dpcm_noise_nrg[g][sfb];	**9**	**uimsbf**
}		
else {		
hcod_sf[dpcm_noise_nrg[g][sfb]];	**1..19**	**vlclbf**
}		
}		
else {		
hcod_sf[dpcm_sf[g][sfb]];	**1..19**	**vlclbf**
}		
}		
}		
}		
}		
}		
else {		
intensity_used = 0;		
noise_used = 0;		
sf_concealment;	**1**	**uimsbf**
rev_global_gain;	**8**	**uimsbf**
length_of_rvlc_sf;	**11/9**	**uimsbf**
for (g = 0; g < num_window_groups; g++) {		
for (sfb=0; sfb < max_sfb; sfb++) {		
if (sfb_cb[g][sfb] != ZERO_HCB) {		
if (is_intensity (g, sfb)) {		
intensity_used = 1;		

```
                    rvlc_cod_sf[dpcm_is_position[g][sfb]];          1..9    vlclbf
                }
                else {
                    if ( is_noise(g,sfb) ) {
                        if ( ! noise_used ) {
                            noise_used = 1;
                            dpcm_noise_nrg[g][sfb];                 9       uimsbf
                        }
                        else {
                            rvlc_cod_sf[dpcm_noise_nrg[g][sfb]];    1..9    vlclbf
                        }
                    }
                    else {
                        rvlc_cod_sf[dpcm_sf[g][sfb]];               1..9    vlclbf
                    }
                }
            }
        }
    }
    if ( intensity_used ) {
        rvlc_cod_sf[dpcm_is_last_position];                         1..9    vlclbf
    }
    noise_used = 0;
    sf_escapes_present;                                             1       uimsbf
    if ( sf_escapes_present ) {
        length_of_rvlc_escapes;                                     8       uimsbf
        for ( g = 0; g < num_window_groups; g++ ) {
            for ( sfb = 0; sfb < max_sfb; sfb++ ) {
                if ( sfb_cb[g][sfb] != ZERO_HCB ) {
                    if ( is_intensity ( g, sfb ) &&
                        dpcm_is_position[g][sfb] == ESC_FLAG ) {
                        rvlc_esc_sf[dpcm_is_position[g][sfb]];      2..20   vlclbf
                    }
                    else {
                        if ( is_noise ( g, sfb ) {
                            if ( ! noise_used ) {
                                noise_used = 1;
                            }
                            else {
                                if (dpcm_noise_nrg[g][sfb] == ESC_FLAG ) {
                                    rvlc_esc_sf[dpcm_noise_nrg[g][sfb]];  2..20   vlclbf
                                }
                            }
                        }
                        else {
                            if (dpcm_sf[g][sfb] == ESC_FLAG ) {
                                rvlc_esc_sf[dpcm_sf[g][sfb]];       2..20   vlclbf
                            }
                        }
                    }
                }
            }
        }
        if ( intensity_used &&
            dpcm_is_last_position == ESC_FLAG ) {
```

	rvlc_esc_sf[dpcm_is_last_position];	**2..20**	**vlclbf**
	}		
}			
if (noise_used) {			
	dpcm_noise_last_position;	**9**	**uimsbf**
}			
}			
}			

Table 4.48 – Syntax of tns_data()

Syntax	No. of bits	Mnemonic
tns_data()		
{		
for (w = 0; w < num_windows; w++) {		
n_filt[w];	**1..2**	**uimsbf**
if (n_filt[w])		
coef_res[w];	**1**	**uimsbf**
for (filt = 0; filt < n_filt[w]; filt++) {		
length[w][filt];	**{4;6}**	**uimsbf**
order[w][filt];	**{3;5}**	**uimsbf**
if (order[w][filt]) {		
direction[w][filt];	**1**	**uimsbf**
coef_compress[w][filt];	**1**	**uimsbf**
for (i = 0; i < order[w][filt]; i++)		
coef[w][filt][i];	**2..4**	**uimsbf**
}		
}		
}		
}		

Table 4.49 – Syntax of ltp_data()

Syntax	No. of bits	Mnemonic
ltp_data()		
{		
if (AudioObjectType == ER_AAC_LD) {		
ltp_lag_update;	**1**	**uimsbf**
if (ltp_lag_update) {		
ltp_lag;	**10**	**uimsbf**
} else {		
ltp_lag = ltp_prev_lag;		
}		
ltp_coef;	**3**	**uimsbf**
for (sfb = 0; sfb < min(max_sfb, MAX_LTP_LONG_SFB); sfb++) {		
ltp_long_used[sfb];	**1**	**uimsbf**
}		
}		
else {		
ltp_lag;	**11**	**uimsbf**
ltp_coef;	**3**	**uimsbf**
if(window_sequence!=EIGHT_SHORT_SEQUENCE) {		
for (sfb=0; sfb<min(max_sfb, MAX_LTP_LONG_SFB); sfb++) {		
ltp_long_used[sfb];	**1**	**uimsbf**
}		

```
        }
    }
}
```

Table 4.50 – Syntax of spectral_data()

Syntax	No. of bits	Mnemonic
Spectral_data()		
{		
for (g = 0; g < num_window_groups; g++) {		
for (i = 0; i < num_sec[g]; i++) {		
if (sect_cb[g][i] != ZERO_HCB &&		
sect_cb[g][i] != NOISE_HCB &&		
sect_cb[g][i] != INTENSITY_HCB &&		
sect_cb[g][i] != INTENSITY_HCB2) {		
for (k = sect_sfb_offset[g][sect_start[g][i]];		
k< sect_sfb_offset[g][sect_end[g][i]];) {		
if (sect_cb[g][i]<FIRST_PAIR_HCB) {		
hcod[sect_cb[g][i]][w][x][y][z];	**1..16**	**vlclbf**
if (unsigned_cb[sect_cb[g][i]])		
quad_sign_bits;	**0..4**	**bslbf**
k += QUAD_LEN;		
}		
else {		
hcod[sect_cb[g][i]][y][z];	**1..15**	**vlclbf**
if (unsigned_cb[sect_cb[g][i]])		
pair_sign_bits;	**0..2**	**bslbf**
k += PAIR_LEN;		
if (sect_cb[g][i] == ESC_HCB) {		
if (y == ESC_FLAG)		
hcod_esc_y;	**5..21**	**vlclbf**
if (z == ESC_FLAG)		
hcod_esc_z;	**5..21**	**vlclbf**
}		
}		
}		
}		
}		
}		
}		

Table 4.51 – Syntax of extension_payload()

Syntax	No. of bits	Mnemonic
extension_payload(cnt)		
{		
extension_type;	**4**	**uimsbf**
align = 4;		
switch(extension_type) {		
case EXT_DYNAMIC_RANGE:		
return dynamic_range_info();		
case EXT_SBR_DATA:		
return sbr_extension_data(id_aac, 0);		Note 1
case EXT_SBR_DATA_CRC:		
return sbr_extension_data(id_aac, 1);		Note 1
case EXT_FILL_DATA:		
fill_nibble; /* must be '0000' */	**4**	**uimsbf**

for (i=0; i<cnt-1; i++) {		
fill_byte[i]; /* must be '10100101' */	**8**	**uimsbf**
}		
return cnt;		
case EXT_DATA_ELEMENT:		
data_element_version;	**4**	**uimsbf**
switch(data_element_version) {		
case ANC_DATA:		
loopCounter = 0;		
dataElementLength = 0;		
do {		
dataElementLengthPart;	**8**	**uimsbf**
dataElementLength += dataElementLengthPart;		
loopCounter++;		
} while (dataElementLengthPart == 255);		
for (i=0; i<dataElementLength; i++) {		
data_element_byte[i];	**8**	**uimsbf**
}		
return (dataElementLength+loopCounter+1);		
default:		
align = 0;		
}		
case EXT_FIL:		
default:		
for (i=0; i<8*(cnt-1)+align; i++) {		
other_bits[i];	**1**	**uimsbf**
}		
return cnt;		
}		
}		
Note 1: id_aac is the id_syn_ele of the corresponding AAC element (ID_SCE or ID_CPE) or ID_SCE in case of CCE.		

Table 4.52 – Syntax of dynamic_range_info()

Syntax	No. of bits	Mnemonic
dynamic_range_info()		
{		
n = 1;		
drc_num_bands = 1;		
pce_tag_present ;	**1**	**uimsbf**
if (pce_tag_present == 1) {		
pce_ instance_tag;	**4**	**uimsbf**
drc_tag_reserved_bits;	**4**	**uimsbf**
n++;		
}		
excluded_chns_present;	**1**	**uimsbf**
if (excluded_chns_present == 1) {		
n += excluded_channels();		
}		
drc_bands_present;	**1**	**uimsbf**
if (drc_bands_present == 1) {		
drc_band_incr;	**4**	**uimsbf**
drc_interpolation_scheme;	**4**	**uimsbf**
n++;		

```
            drc_num_bands = drc_num_bands + drc_band_incr;
            for (i = 0; i < drc_num_bands; i++) {
                drc_band_top[i];                                    8       uimsbf
                n++;
            }
        }
        prog_ref_level_present;                                     1       uimsbf
        if (prog_ref_level_present == 1) {
            prog_ref_level;                                         7       uimsbf
            prog_ref_level_reserved_bits;                           1       uimsbf
            n++;
        }
        for (i = 0; i < drc_num_bands; I++) {
            dyn_rng_sgn[i];                                         1       uimsbf
            dyn_rng_ctl[i];                                         7       uimsbf
            n++;
        }
        return n;
}
```

Table 4.53 – Syntax of excluded_channels()

Syntax	No. of bits	Mnemonic
Excluded_channels( )		
{		
n = 0;		
num_excl_chan = 7;		
for (i = 0; i < 7; i++)		
exclude_mask[i];	**1**	**uimsbf**
n++;		
while (**additional_excluded_chns[n-1]** == 1) {	**1**	**uimsbf**
for (i = num_excl_chan; i < num_excl_chan+7; i++)		
exclude_mask[I];	**1**	**uimsbf**
n++;		
num_excl_chan += 7;		
}		
return n;		
}		

Table 4.54 – Syntax of ms_data()

Syntax	No. of bits	Mnemonic
ms_data()		
{		
for (g = 0; g < num_window_groups; g++) {		
for (sfb = last_max_sfb_ms; sfb < max_sfb; sfb++) {		
ms_used[g][sfb];	**1**	**bslbf**
}		
}		
}		

4.4.2.8 Payloads for the audio object type SBR

Table 4.55 – Syntax of sbr_extension_data()

Syntax	No. of bits	Mnemonic
sbr_extension_data(id_aac, crc_flag)		

{		
num_sbr_bits = 0;		
if (crc_flag) {		
bs_sbr_crc_bits;	**10**	**uimsbf**
num_sbr_bits += 10;		
}		
if (sbr_layer != SBR_STEREO_ENHANCE) {		Note 1
num_sbr_bits += 1;		
if (**bs_header_flag**)	**1**	
num_sbr_bits += sbr_header();		Note 2
}		
num_sbr_bits += sbr_data(id_aac, bs_amp_res);		Note 2
num_align_bits = (8*cnt - 4 - num_sbr_bits)%8;		
bs_fill_bits;	num_align_bits	**uimsbf**
return ((num_sbr_bits + num_align_bits + 4) / 8)		
}		

Note 1: When the SBR tool is used with a non-scalable AAC core coder, the value of the helper variable sbr_layer is SBR_NOT_SCALABLE. When the SBR tool is used with a scalable AAC core coder, the value of the helper variable sbr_layer depends on the current layer and the scalability configuration of the AAC core coder as defined in Table 4.104 in Subclause 4.5.2.8.2.4.
Note 2: sbr_header() and sbr_data() return the number of bits read (cnt is a parameter in extension_payload()).

Table 4.56 – Syntax of sbr_header()

Syntax	No. of bits	Mnemonic
sbr_header()		
{		
bs_amp_res;	**1**	
bs_start_freq;	**4**	**uimsbf**, Note 1
bs_stop_freq;	**4**	**uimsbf**, Note 1
bs_xover_band;	**3**	**uimsbf**, Note 2
bs_reserved;	**2**	**uimsbf**
bs_header_extra_1;	**1**	
bs_header_extra_2;	**1**	
if (bs_header_extra_1) {		Note 3
bs_freq_scale;	**2**	**uimsbf**
bs_alter_scale;	**1**	
bs_noise_bands;	**2**	**uimsbf**
}		
if (bs_header_extra_2) {		Note 3
bs_limiter_bands;	**2**	**uimsbf**
bs_limiter_gains;	**2**	**uimsbf**
bs_interpol_freq;	**1**	

bs_smoothing_mode;	**1**	
}		
}		
Note 1: bs_start_freq and bs_stop_freq shall define a frequency band that does not exceed the limits defined in subclause 4.6.18.3.6. Note 2: Index to the master frequency band table, indicating where the current SBR range begins Note 3: If this bit is not set the default values for the underlying data elements shall be used disregarded any previous value.		

Table 4.57 – Syntax of sbr_data()

Syntax	No. of bits	Mnemonic

sbr_data(id_aac, bs_amp_res)
{
switch (sbr_layer) {
case SBR_NOT_SCALABLE
switch (id_aac) {
case ID_SCE
sbr_single_channel_element(bs_amp_res)
break;
case ID_CPE
sbr_channel_pair_element(bs_amp_res)
break;
}
break;
case SBR_MONO_BASE
sbr_channel_pair_base_element(bs_amp_res)
break;
case SBR_STEREO_ENHANCE
sbr_channel_pair_enhance_element(bs_amp_res)
break;
case SBR_STEREO_BASE
sbr_channel_pair_element(bs_amp_res)
break;
}
}
		Note 1

Note 1: When the SBR tool is used with a non-scalable AAC core coder, the value of the helper variable sbr_layer is SBR_NOT_SCALABLE. When the SBR tool is used with a scalable AAC core coder, the value of the helper variable sbr_layer depends on the current layer and the scalability configuration of the AAC core coder as defined in Table 4.104 in Subclause 4.5.2.8.2.4.		

Table 4.58 – Syntax of sbr_single_channel_element()

Syntax	No. of bits	Mnemonic

sbr_single_channel_element(bs_amp_res)
{
if (**bs_data_extra**)
bs_reserved;
 	
sbr_grid(0);
sbr_dtdf(0);
sbr_invf(0);
sbr_envelope(0, 0, bs_amp_res);
sbr_noise(0, 0);
	1
	4
		uimsbf

if (**bs_add_harmonic_flag**[0])	**1**	
sbr_sinusoidal_coding(0);		
if (**bs_extended_data**) {	**1**	
cnt = **bs_extension_size**;	**4**	**uimsbf**
if (cnt == 15)		
cnt += **bs_esc_count**;	**8**	**uimsbf**
num_bits_left = 8 * cnt;		
while (num_bits_left > 7) {		
bs_extension_id;	**2**	**uimsbf**
num_bits_left -= 2;		
sbr_extension(bs_extension_id, num_bits_left);		Note 1
}		
bs_fill_bits;	num_bits _left	
}		
}		

Note 1: sbr_extension() shall decrease the variable num_bits_left by the number of bits read from the bitstream payload within sbr_extension(). The sbr_extension() element is reserved for future use.

Table 4.59 – Syntax of sbr_channel_pair_element()

Syntax	No. of bits	Mnemonic
sbr_channel_pair_element(bs_amp_res)		
{		
if (**bs_data_extra**) {	**1**	
bs_reserved;	**4**	**uimsbf**
bs_reserved;	**4**	**uimsbf**
}		
if (**bs_coupling**) {	**1**	
sbr_grid(0);		
sbr_dtdf(0);		
sbr_dtdf(1);		
sbr_invf(0);		
sbr_envelope(0,1, bs_amp_res);		
sbr_noise(0,1);		
sbr_envelope(1,1, bs_amp_res);		
sbr_noise(1,1);		
} else {		
sbr_grid(0);		
sbr_grid(1);		
sbr_dtdf(0);		
sbr_dtdf(1);		
sbr_invf(0);		
sbr_invf(1);		
sbr_envelope(0,0, bs_amp_res);		
sbr_envelope(1,0, bs_amp_res);		
sbr_noise(0,0);		
sbr_noise(1,0);		
}		

if (**bs_add_harmonic_flag**[0])	**1**	
sbr_sinusoidal_coding(0);		
if (**bs_add_harmonic_flag**[1])	**1**	
sbr_sinusoidal_coding(1);		
if (**bs_extended_data**) {	**1**	
cnt = **bs_extension_size**;	**4**	**uimsbf**
if (cnt == 15)		
cnt += **bs_esc_count**;	**8**	**uimsbf**
num_bits_left = 8 * cnt;		
while (num_bits_left > 7) {		
bs_extension_id;	**2**	**uimsbf**
num_bits_left -= 2;		
sbr_extension(bs_extension_id, num_bits_left);		Note 1
}		
bs_fill_bits;	num_bits_left	
}		
}		

Note 1: sbr_extension() shall decrease the variable num_bits_left by the number of bits read from the bitstream payload within sbr_extension(). The sbr_extension() element is reserved for future use.

Table 4.60 – Syntax of sbr_channel_pair_base_element()

Syntax	No. of bits	Mnemonic
sbr_channel_pair_base_element(bs_amp_res)		
{		
if (**bs_data_extra**) {	**1**	
bs_reserved;	**4**	**uimsbf**
bs_reserved;	**4**	**uimsbf**
}		
bs_coupling	**1**	Note 1
sbr_grid(0);		
sbr_dtdf(0);		
sbr_invf(0);		
sbr_envelope(0,1, bs_amp_res);		
sbr_noise(0,1);		
if (**bs_add_harmonic_flag**[0])	**1**	
sbr_sinusoidal_coding(0);		
if (**bs_extended_data**) {	**1**	
cnt = **bs_extension_size**;	**4**	**uimsbf**
if (cnt == 15)		
cnt += **bs_esc_count**;	**8**	**uimsbf**
num_bits_left = 8 * cnt;		
while (num_bits_left > 7) {		
bs_extension_id;	**2**	**uimsbf**
num_bits_left -= 2;		
sbr_extension(bs_extension_id, num_bits_left);		Note 2

}		
bs_fill_bits;	num_bits _left	
}		
}		
Note 1: bs_coupling shall have the value 1.		
Note 2: sbr_extension() shall decrease the variable num_bits_left by the number of bits read from the bitstream payload within sbr_extension(). The sbr_extension() element is reserved for future use.		

Table 4.61 – Syntax of sbr_channel_pair_enhance_element()

Syntax	No. of bits	Mnemonic
sbr_channel_pair_enhance_element(bs_amp_res)		
{		
sbr_dtdf(1);		
sbr_envelope(1,1, bs_amp_res);		
sbr_noise(1,1);		
if (**bs_add_harmonic_flag**[1])	**1**	
sbr_sinusoidal_coding(1);		
}		

Table 4.62 – Syntax of sbr_grid()

Syntax	No. of bits	Mnemonic
sbr_grid(ch)		
{		
switch (**bs_frame_class**) {	**2**	**uimsbf**
case FIXFIX		
bs_num_env[ch] = 2^ **tmp**;	**2**	**uimsbf**, Note 1
if (bs_num_env[ch] == 1)		
bs_amp_res = 0;		
bs_freq_res[ch][0];	**1**	
for (env = 1; env < bs_num_env[ch]; env++)		
bs_freq_res[ch][env] = bs_freq_res[ch][0];		
break;		
case FIXVAR		
bs_var_bord_1[ch];	**2**	**uimsbf**
bs_num_env[ch] = **bs_num_rel_1**[ch] + 1;	**2**	**uimsbf**
for (rel = 0; rel < bs_num_env[ch]-1; rel++)		
bs_rel_bord_1[ch][rel] = 2* **tmp** + 2;	**2**	**uimsbf**
ptr_bits = ceil (log (bs_num_env[ch] + 1) / log (2));		Note 2
bs_pointer[ch];	ptr_bits	**uimsbf**
for (env = 0; env < bs_num_env[ch]; env++)		
bs_freq_res[ch][bs_num_env[ch] – 1 – env];	**1**	
break;		
case VARFIX		
bs_var_bord_0[ch];	**2**	**uimsbf**
bs_num_env[ch] = **bs_num_rel_0**[ch] + 1;	**2**	**uimsbf**
for (rel = 0; rel < bs_num_env[ch]-1; rel++)		
bs_rel_bord_0[ch][rel] = 2* **tmp** + 2;	**2**	**uimsbf**
ptr_bits = ceil (log (bs_num_env[ch] + 1) / log (2));		Note 2
bs_pointer[ch];	ptr_bits	**uimsbf**

for (env = 0; env < bs_num_env[ch]; env++)		
bs_freq_res[ch] [env];	**1**	
break;		
case VARVAR		
bs_var_bord_0[ch];	**2**	**uimsbf**
bs_var_bord_1[ch];	**2**	**uimsbf**
bs_num_rel_0[ch];	**2**	**uimsbf**
bs_num_rel_1[ch];	**2**	**uimsbf**
bs_num_env[ch] = bs_num_rel_0[ch] + bs_num_rel_1[ch] + 1;		Note 1
for (rel = 0; rel < bs_num_rel_0[ch]; rel++)		
bs_rel_bord_0[ch][rel] = 2* **tmp** + 2;	**2**	**uimsbf**
for (rel = 0; rel < bs_num_rel_1[ch]; rel++)		
bs_rel_bord_1[ch][rel] = 2* **tmp** + 2;	**2**	**uimsbf**
ptr_bits = ceil (log(bs_num_env[ch] + 1) / log (2));		Note 2
bs_pointer[ch];	ptr_bits	**uimsbf**
for (env = 0; env < bs_num_env[ch]; env++)		
bs_freq_res[ch][env];	**1**	
break;		
}		
if (bs_num_env[ch] > 1)		
bs_num_noise[ch] = 2;		
else		
bs_num_noise[ch] = 1;		
}		
Note 1: bs_num_env is restricted according to subclause 4.6.18.3.6.		
Note 2: the division (/) is a float division without rounding or truncation.		

Table 4.63 – Syntax of sbr_dtdf()

Syntax	No. of bits	Mnemonic
sbr_dtdf(ch)		
{		
for (env = 0; env < bs_num_env[ch]; env++)		
bs_df_env[ch][env];	**1**	
for (noise = 0; noise < bs_num_noise[ch]; noise++)		
bs_df_noise[ch][noise];	**1**	
}		

Table 4.64 – Syntax of sbr_invf()

Syntax	No. of bits	Mnemonic
sbr_invf(ch)		
{		
for (n = 0; n<num_noise_bands[ch]; n++)		Note 1
bs_invf_mode[ch][n];	**2**	**uimsbf**
}		
Note 1: num_noise_bands[ch] is derived from the header, according to subclause 4.6.18.3 and is named N_Q.		

Table 4.65 – Syntax of sbr_envelope()

Syntax	No. of bits	Mnemonic
sbr_envelope(ch, bs_coupling, bs_amp_res)		
{		

```
    if (bs_coupling) {
        if (ch) {
            if (bs_amp_res) {
                t_huff = t_huffman_env_bal_3_0dB;
                f_huff = f_huffman_env_bal_3_0dB;
            } else {
                t_huff = t_huffman_env_bal_1_5dB;
                f_huff = f_huffman_env_bal_1_5dB;
            }
        } else {
            if (bs_amp_res) {
                t_huff = t_huffman_env_3_0dB;
                f_huff = f_huffman_env_3_0dB;
            } else {
                t_huff = t_huffman_env_1_5dB;
                f_huff = f_huffman_env_1_5dB;
            }
        }
    } else {
        if (bs_amp_res) {
            t_huff = t_huffman_env_3_0dB;
            f_huff = f_huffman_env_3_0dB;
        } else {
            t_huff = t_huffman_env_1_5dB;
            f_huff = f_huffman_env_1_5dB;
        }
    }

    for (env = 0; env < bs_num_env[ch]; env++) {
        if (bs_df_env[ch][env] == 0) {
            if (bs_coupling && ch) {
                if (bs_amp_res)
                    bs_data_env[ch][env][0] = bs_env_start_value_balance;      5        uimsbf
                else
                    bs_data_env[ch][env][0] = bs_env_start_value_balance;      6        uimsbf
            } else {
                if (bs_amp_res)
                    bs_data_env[ch][env][0] = bs_env_start_value_level;        6        uimsbf
                else
                    bs_data_env[ch][env][0] = bs_env_start_value_level;        7        uimsbf
            }
            for (band = 1;  band < num_env_bands[bs_freq_res[ch][env]]; band++)        Note 1
                bs_data_env[ch][env][band] = sbr_huff_dec(f_huff, bs_codeword);  1..18  Note 2
        } else {
            for (band = 0; band < num_env_bands[bs_freq_res[ch][env]]; band++)        Note 1
                bs_data_env[ch][env][band] = sbr_huff_dec(t_huff, bs_codeword);  1..18  Note 2
        }
    }
}
```

Note 1: num_env_bands[bs_freq_res[ch][env]] is derived from the header according to subclause 4.6.18.3 and is named **n**.
Note 2: sbr_huff_dec() is defined in Annex 4.A.6.1.

Table 4.66 – Syntax of sbr_noise()

Syntax	No. of bits	Mnemonic
sbr_noise(ch,bs_coupling) {		
if (bs_coupling) {		
if (ch) {		
t_huff = t_huffman_noise_bal_3_0dB;		
f_huff = f_huffman_noise_bal_3_0dB;		
} else {		
t_huff = t_huffman_noise_3_0dB;		
f_huff = f_huffman_noise_3_0dB;		
}		
} else {		
t_huff = t_huffman_noise_3_0dB;		
f_huff = f_huffman_noise_3_0dB;		
}		
for (noise = 0; noise < bs_num_noise[ch]; noise++) {		
if (bs_df_noise[ch][noise] == 0) {		
if (bs_coupling && ch)		
bs_data_noise[ch][noise][0] = **bs_noise_start_value_balance**;	**5**	**uimsbf**
else		
bs_data_noise[ch][noise][0] = **bs_noise_start_value_level**;	**5**	**uimsbf**
for (band = 1; band < num_noise_bands[ch]; band++)		Note 1
bs_data_noise[ch][noise][band] = sbr_huff_dec(f_huff,**bs_codeword**);	**1..18**	Note 2
} else {		
for (band = 0; band < num_noise_bands[ch]; band++)		Note 1
bs_data_noise[ch][noise][band] = sbr_huff_dec(t_huff,**bs_codeword**);	**1..18**	Note 2
}		
}		
}		
Note 1: num_noise_bands[ch] is derived from the header according to subclause 4.6.18.3 and is named N_Q.		
Note 2: sbr_huff_dec() is defined in Annex 4.A.6.1.		

Table 4.67 – Syntax of sbr_sinusoidal_coding()

Syntax	No. of bits	Mnemonic
sbr_sinusoidal_coding(ch)		
{		
for (n = 0; n<num_high_res[ch]; n++)		Note 1
bs_add_harmonic[ch][n]	**1**	
}		
Note 1: num_high_res[ch] is derived from the header according to subclause 4.6.18.3 and is named N_{High}.		

4.5 Overall data structure

4.5.1 Decoding of the GA specific configuration

4.5.1.1 GASpecificConfig()

The call parameters 'samplingFrequencyIndex', 'channelConfiguration', 'audioObjectType' are passed down from the audio specific configuration element defined in subpart 1. The information contained in these parameters is mandatory for the decoding process.

If the sampling rate is not one of the rates listed in the right column in Table 4.68, the sampling frequency dependent tables (code tables, scale factor band tables etc.) must be deduced in order for the bitstream payload to be parsed. Since a given sampling frequency is associated with only one sampling frequency table, and since maximum flexibility is desired in the range of possible sampling frequencies, the following table shall be used to associate an implied sampling frequency with the desired sampling frequency dependent tables.

Table 4.68 – Sampling frequency mapping

Frequency range (in Hz)	Use tables for sampling frequency (in Hz)
f >= 92017	96000
92017 > f >= 75132	88200
75132 > f >= 55426	64000
55426 > f >= 46009	48000
46009 > f >= 37566	44100
37566 > f >= 27713	32000
27713 > f >= 23004	24000
23004 > f >= 18783	22050
18783 > f >= 13856	16000
13856 > f >= 11502	12000
11502 > f >= 9391	11025
9391 > f	8000

If a certain sampling frequency dependent table stated in the right column of Table 4.68 is not defined, the nearest defined table shall be used.

frameLengthFlag Length of the frame, number of spectral lines, respective.
For all General Audio Object Types except AAC SSR and ER AAC LD: If set to "0" a 1024/128 lines IMDCT is used and frameLength is set to 1024, if set to "1" a 960/120 line IMDCT is used and frameLength is set to 960.
For ER AAC LD: If set to "0" a 512 lines IMDCT is used and frameLength is set to 512, if set to "1" a 480 line IMDCT is used and frameLength is set to 480.
For AAC SSR: Must be set to "0". A 256/32 lines IMDCT is used.
Note: The actual number of lines for the IMDCT (first or second value) is distinguished by the value of window_sequence.

DependsOnCoreCoder Signals that a core coder has been used in an underlying base layer of a scalable AAC configuration.

CoreCoderDelay The delay in samples that has to be applied to the up-sampled (if necessary) core decoder output, before the MDCT calculation.

extensionFlag: Shall be '0' for audio object types 1, 2, 3, 4, 6, 7. Shall be '1' for audio object types 17, 19, 20, 21, 22, 23.

layerNr: A 3-bit field indicating the AAC layer number in a scalable configuration. The first AAC layer is signaled by a value of 0.

numOfSubFrame A 5-bit unsigned integer value representing the number of the sub-frames which are grouped and transmitted in a super-frame.

layer_length An 11-bit unsigned integer value representing the average length of the large-step layers in bytes.

aacSectionDataResilienceFlag This flag signals a different coding scheme of AAC section data. If codebook 11 is used, this scheme transmits additional information about the maximum absolute value for spectral lines. This allows error detection of spectral lines that are larger than this value.

aacScalefactorDataResilienceFlag This flag signals a different coding scheme of the AAC scalefactor data, that is more resilient against errors as the original one.

aacSpectralDataResilienceFlag This flag signals a different coding scheme (HCR) of the AAC spectral data, that is more resilient against errors as the original one

extensionFlag3 Extension flag for the future use. Shall be '0'.

Restrictions:

program_config_element() shall be used only for the audio object types AAC main, AAC SSR, AAC LC and AAC LTP.

4.5.1.2 Program config element (PCE)

See ISO/IEC13818-7 (13818-7:2005, subclause 8.5 "Program Config Element (PCE)").

The following changes apply in the context of MPEG-4:

A program_config_element() may occur outside the AAC payload e. g. as part of the GASpecificConfig() or the adif_header(), but also inside the AAC payload as syntactic element in a raw_data_block().

Note that the channel configuration given in a program_config_element() inside the AAC payload is evaluated only, if no channel configuration is given outside the AAC payload. In the context of ISO/IEC 14496-3 this is only the case for MPEG-4 ADTS with channel_configation==0.

In any case only one program may be configured at a certain time.

object_type The two-bit object type index from Table 4.69. This field replaces the profile field of the PCE in ISO/IEC 13818-7 in a backward compatible way.

Table 4.69 – Object type index

Index	object type
0	AAC Main
1	AAC LC
2	AAC SSR
3	AAC LTP

sampling_frequency_index Indicates the sampling frequency of the program according to the table defined in subclause 1.6.3.3 (samplingFrequencyIndes). The escape value is not permitted.

4.5.1.2.1 Channel configuration

The AAC audio syntax provides three ways to convey the mapping of channels within a set of syntactic elements to physical locations of speakers. However in in the context of ISO/IEC 14496-3 only two of them are permitted.

4.5.1.2.1.1 Explicit channel mapping using default channel settings

Default channel mappings are defined in subpart 1, Table 1.17 (values larger 0). If MPEG-4 Audio is used together with the MPEG-4 Systems audio compositor only these mappings shall be used.

4.5.1.2.1.2 Explicit channel mapping using a program_config_element()

Any possible channel configuration can be specified using a program_config_element().The same specifications and restrictions as defined in ISO/IEC 13818-7 apply with respect to the PCE when used in the context of ISO/IEC 14496-3.

An MPEG-4 decoder is always required to parse any program_config_element() inside the AAC payload. However, the decoder is only required to evaluate it, if no channel configuration is given outside the AAC payload.

4.5.1.2.1.3 Implicit channel mapping

This kind of channel mapping as specified in ISO/IEC13818-7 is not permitted in the context of ISO/IEC 14496-3.

4.5.1.2.2 Matrix-mixdown method

4.5.1.2.2.1 Description

The matrix-mixdown method applies only for mixing a 3-front/2-back speaker configuration, 5-channel program, down to a stereo or a mono program. It is not applicable to any program with other than the 3/2 configuration.

4.5.1.2.2.2 Matrix-mixdown process

A derived stereo signal can be generated within a matrix-mixdown decoder by use of one of the two following sets of equations.

Set 1:

$$L' = \frac{1}{1+1/\sqrt{2}+A} \cdot [L + C/\sqrt{2} + A \cdot L_S]$$

$$R' = \frac{1}{1+1/\sqrt{2}+A} \cdot [R + C/\sqrt{2} + A \cdot R_S]$$

Set 2:

$$L' = \frac{1}{1+1/\sqrt{2}+2 \cdot A} \cdot [L + C/\sqrt{2} - A \cdot (L_S + R_S)]$$

$$R' = \frac{1}{1+1/\sqrt{2}+2 \cdot A} \cdot [R + C/\sqrt{2} + A \cdot (L_S + R_S)]$$

Where L, C, R, LS and RS are the source signals, L' and R' are the derived stereo signals and A is the matrix coefficient indicated by matrix_mixdown_idx. LFE channels are omitted from the mixdown.

If pseudo_surround_enable is not set, then only set 1 should be used. If pseudo_surround_enable is set, then either set 1 or set 2 equations can be used, depending on whether the receiver has facilities to invoke some form of surround synthesis.

As further information it should be noted that one can derive a mono signal using the following equation:

$$M = \frac{1}{3+2 \cdot A} \cdot \left[L + C + R + A \cdot (L_S + R_S)\right]$$

4.5.1.2.2.3 Advisory

The matrix-mixdown provision enables a mode of operation which may be beneficial in some circumstances. However, it is advised that this method should not be used. The psychoacoustic principles on which the audio coding is based are violated by this form of post-processing, and a perceptually faithful reconstruction of the signal cannot be guaranteed. The preferred method is to use the stereo or mono mixdown channels in the AAC syntax to provide stereo or mono programming which is specifically created by conventional studio mixing prior to bitrate reduction.

The stereo and mono mixdown channels additionally enable the content provider to separately optimize the stereo and multichannel program mixes - this is not possible by using the matrix-mixdown method.

It is additionally relevant to note that, due to the algorithms used for the multichannel and stereo mixdown coding, a better combination of quality and bitrate is usually provided by use of the stereo mixdown channels than can be provided by the matrix-mixdown process.

4.5.1.2.2.4 Tables

Table 4.70 – Matrix-mixdown coefficients

matrix_mixdown_idx	A
0	$1/\sqrt{2}$
1	$1/2$
2	$1/(2\sqrt{2})$
3	0

4.5.2 Decoding of the GA bitstream payloads

4.5.2.1 Top level payloads for the audio object types AAC main, AAC SSR, AAC LC and AAC LTP

4.5.2.1.1 Definitions

raw_data_block() — block of raw data that contains audio data for a time period of 1024 or 960 samples, related information and other data. There are seven syntactic elements, identified by the data element id_syn_ele. The audio_channel_element()'s in one raw_data_block() must have one and only one sampling rate. In the raw_data_block(), several instances of the same syntactic element may occur, but must have a different 4 bit element_instance_tag, except for data_stream_element()'s and fill_element()'s. Therefore, in one raw_data_block(), there can be from 0 to at most 16 instances of any syntactic element,except for data_stream_element()'s and fill_element()'s, where this limitation does not apply. If multiple data_stream_element()'s occur which have the same element_instance_tag then they are part of the same data stream. The fill_element() has no element_instance_tag (since the content does not require subsequent reference) and can occur any number of times. The end of a raw_data_block() is indicated with a special id_syn_ele (TERM), which may occur only once in a raw_data_block().

id_syn_ele — a data element that identifies one of the following syntactic elements:

Table 4.71 – Syntactic elements

ID name	encoding	Abbreviation	Syntactic Element
ID_SCE	0x0	SCE	single_channel_element()
ID_CPE	0x1	CPE	channel_pair_element()
ID_CCE	0x2	CCE	coupling_channel_element()
ID_LFE	0x3	LFE	lfe_channel_element()
ID_DSE	0x4	DSE	data_stream_element()
ID_PCE	0x5	PCE	program_config_element()
ID_FIL	0x6	FIL	fill_element()
ID_END	0x7	TERM	

single_channel_element() — abbreviaton SCE. Syntactic element of the bitstream containing coded data for a single audio channel. A single_channel_element() basically consists of an individual_channel_stream(). There may be up to 16 such elements per raw data block, each one must have a unique element_instance_tag.

channel_pair_element() — abbreviation CPE. Syntactic element of the bitstream payload containing data for a pair of channels. A channel_pair_element() consists of two individual_channel_streams and additional joint channel coding information. The two channels may share common side information. The channel_pair_element() has the same restrictions as the single channel element as far as element_instance_tag, and number of occurrences.

coupling_channel_element()	Abbreviation CCE. Syntactic element that contains audio data for a coupling channel. A coupling channel represents the information for multi-channel intensity for one block, or alternately for dialogue for multilingual programming. The rules for number of coupling_channel_element()'s and instance tags are as for single_channel_element().
lfe_channel_element()	Abbreviation LFE. Syntactic element that contains a low sampling frequency enhancement channel. The rules for the number of lfe_channel_element()'s and instance tags are as for single_channel_element()'s.
program_config_element()	Abbreviation PCE. Syntactic element that contains program configuration data. The rules for the number of program_config_element()'s and element instance tags are the same as for single_channel_element()'s. PCEs must come before all other syntactic elements in a raw_data_block().
fill_element()	Abbreviation FIL. Syntactic element that contains fill data. There may be any number of fill elements, that can come in any order in the raw data block.
data_stream_element()	Abbreviation DSE. Syntactic element that contains data. Again, there are 16 element_instance_tags. There is, however, no restriction on the number of data_stream_element()'s with any one instance tag, as a single data stream may continue across multiple data_stream_element()'s with the same instance tag.
element_instance_tag	Unique instance tag for syntactic elements other than fill_element(). All syntactic elements containing instance tags may occur more than once, but, except for data_stream_element()'s, must have a unique element_instance_tag in each raw_data_block(). This tag is also used to reference audio syntactic elements in single_channel_element()'s, channel_pair_element()'s, lfe_channel_element()'s, data_channel_element()'s, and coupling_channel_element()'s inside a program_config_element(), and provides the possibility of up to 16 independent program_config_element()'s.
audio_channel_element	generic term for single_channel_element(), channel_pair_element, coupling_channel_element() and lfe_channel_element().
common_window	a flag indicating whether the two individual_channel_streams share a common ics_info or not. In case of sharing, the ics_info is part of the channel_pair_element() and must be used for both channels. Otherwise, the ics_info is part of each individual_channel_stream.

4.5.2.1.2 Decoding process

Assuming that the start of a raw_data_block is known, it can be decoded without any additional „transport-level" information exept the sampling rate (needed for the selection of sfb and other tables) and produces 1024 or 960 audio samples per output channel.

Table 4.72 – Examples of the simplest possible bitstream payloads

bitstream payload segment	output signal
<SCE><TERM><SCE><TERM>…	mono signal
<CPE><TERM><CPE><TERM>…	stereo signal
<SCE><CPE><CPE><LFE><TERM><SCE><CPE><CPE><LFE><TERM>…	5.1 channel signal

where angle brackets (< >) are used to delimit syntactic elements. For the mono signal each SCE must have the same value in its **element_instance_tag**, and similarly, for the stereo signal each CPE must have the same value in its **element_instance_tag**. For the 5.1 channel signal each SCE must have the same value in its **element_instance_tag**, each CPE associated with the front channel pair must have the same value in its **element_instance_tag**, and each CPE associated with the back channel pair must have the same value in its **element_instance_tag**.

If these bitstream payloads are to be transmitted over a constant rate channel then they might include a fill_element() to adjust the instantaneous bit rate. In this case an example of a coded stereo signal is

<CPE><FIL><TERM><CPE><FIL><TERM>…

If the bitstream payloads are to carry ancillary data and run over a constant rate channel then an example of a coded stereo signal is

<CPE><DSE><FIL><TERM><CPE><DSE><FIL><TERM>…

All data_stream_element()'s have the same element_instance_tag if they are part of the same data stream.

A single_channel_element() is composed of an element_instance_tag and an individual_channel_stream. In this case ics_info is always located in the individual_channel_stream.

A channel_pair_element() begins with an element_instance_tag and common_window flag. If the common_window equals '1', then ics_info is shared amongst the two individual_channel_stream elements and the MS information is transmitted. If common_window equals '0', then there is an ics_info within each individual_channel_stream and there is no MS information.

4.5.2.1.3 Low frequency enhancement (LFE) channel element

4.5.2.1.3.1 General

In order to maintain a regular structure in the decoder, the lfe_channel_element() is defined as a standard individual_channel_stream(0) element, i.e. equal to a single_channel_element(). Thus, decoding can be done using the standard procedure for decoding a single_channel_element().

In order to accommodate a more bitrate and hardware efficient implementation of the LFE decoder, however, several restrictions apply to the options used for the encoding of this element:

- The window_shape field is always set to 0, i.e. sine window
- The window_sequence field is always set to 0 (ONLY_LONG_SEQUENCE)
- Only the lowest 12 spectral coefficients of any LFE may be non-zero
- No Temporal Noise Shaping is used, i.e. tns_data_present is set to 0
- No prediction is used, i.e. predictor_data_present is set to 0

4.5.2.1.4 Data stream element (DSE)

See ISO/IEC13818-7 (13818-7:2005, subclause 8.6 "Data Stream Element (DSE)").

4.5.2.1.5 Fill element (FIL)

4.5.2.1.5.1 Data elements

count	Initial value for length of fill data.
esc_count	Incremental value of length of fill data.

4.5.2.1.5.2 Helper elements

cnt	value equal to the total length in bytes of all subsequent extensioin_payload()'s.

Any number of fill elements is allowed.

Fill elements containing an extension_payload() with an extension_type of EXT_SBR_DATA or EXT_SBR_DATA_CRC shall not contain any other extension_payload of any other extension_type.

4.5.2.1.5.3 Decoding process

The syntactic element **count** gives the initial value of the length of the subsequent extension_payload()'s. In the same way as for the data element this value is incremented with the value of **esc_count** if **count** equals 15. The resulting number (cnt) gives the number of bytes to be read.

4.5.2.2 Payloads for the audio object type AAC scalable

4.5.2.2.1 Definitions

aac_scalable_main_element()	abbreviation ASME; syntactic element of the bitstream payload containing coded data for the first AAC coding layer in a scalable configuration. This type of syntax can be used also for single coding layer (non-scalable) applications. In this case exactly one ASME contains all the coded information. An ASME consists of an aac_scalable_main_header(), and for each coded audio output channel, one individual_channel_stream(). The maximum number of such elements is limited to 1 for each audio object.
aac_scalable_main_header()	contains all side information required for the first AAC coding layer, with the exception of the side information which is transmitted within the individual channel streams. The ics_info() sub-block of AAC is not used here. Instead the information, normally carried in ics_info(), is contained directly in this element.
aac_scalable_extension_element()	abbreviation ASEE; syntactic element of the bitstream payload containing coded data for all other than the first AAC coding layer in a scalable configuration. An ASEE consists of an aac_scalable_extension_header(), and for each coded audio output channel, one individual_channel_stream(). The maximum number of such elements is limited to 7 for one audio object.
aac_scalable_extension_header()	contains all side information required for an AAC extension layer, with the exception of the side information that is transmitted within the individual channel streams. Extension in this context means, that this is not the first AAC coding layer.
bits_to_decode()	a helper function; returns the number of bits not yet decoded in the current top level payload if the length of that payload is signaled by a system/transport layer. If the length of the top level payload is unknown, bits_to_decode() returns 0.
diff_control[w][dc_group]	for each window this is the FSS control information for one dc_group. If the window type is not SHORT_WINDOW. **diff_control[w][dc_group]** is huffman encoded using Table 4.146 in the bitstream payload.
diff_control_lr[w][sfb]	element used in a mono GA / stereo GA configuration, to control the interaction of the M-channel with the L and R channel.

4.5.2.2.1.1 Help elements

this_layer_stereo	set to ‘1’, if the current layer is a stereo layer; set to ‘0’ otherwise.
mono_layer_flag	set to ‘1’, if there is any mono layer; set to ‘0’ otherwise.
mono_stereo_flag	set to ‘1’, if there is at least one mono layer and this is the first stereo layer; set to ‘0’ otherwise.
last_max_sfb	max_sfb of the previous coding layer. If the previous layer is running at a different sampling rate or is a non GA coder, last_max_sfb is set to ‘4* no_of_dc_groups-1’ if the window type is not SHORT_WINDOW, otherwise it is set to the lowest sfb covering all diff_short_lines.
last_max_sfb_ms	max_sfb of the previous stereo layer; set to ‘0’, if the previous layer is a mono layer
last_mono_max_sfb	max_sfb of the hightst mono layer.
core_flag	set to ‘1’, if a core coder is present. Within the MPEG-4 Audio standard the only valid core coder is MPEG-4 Celp, although the mechanisms could be used e.g. for other waveform coders or an AAC coder operating at a lower sampling rate than the AAC extension layer; set to ‘0’ if there is no core coder.

tvq_layer_present	set to ‘1’, if a TwinVQ coder is present in the previous layer; set to ‘0’ otherwise.
tvq_tns_present	set to ‘1’, if tns_present is set to 1 in the previous TwinVQ layer; set to ‘0’ otherwise.
tvq_stereo	set to ‘1’, if the previous TwinVQ layer is stereo; set to ‘0’ otherwise.
tvq_mono_tns	set to ‘1’, if there is a tvq mono layer present that uses tns; set to 0 if there is a tvq mono layer present that does not use tns.
tvq_stereo_tns_left	set to ‘1’, if there is a tvq stereo layer present that uses tns in the left channel; set to 0 if there is a tvq stereo layer present that does not use tns in the left channel.
tvq_stereo_tns_right	set to ‘1’, if there is a tvq stereo layer present that uses tns in the right channel; set to 0 if there is a tvq stereo layer present that does not use tns in the right channel.
tns_aac_tvq_en[ch]	set to ‘1’ if a tns_data_present bit is transmitted in an ASME, if a TwinVQ layer is present (see Table 4.76).

4.5.2.2.2 Decoding process

Assuming that the start of an **ASME** or **ASEE** is known, it can be decoded with the additional information given by the GASpecificConfig(). 1024 or 960 audio samples per output channel are produced for each element.

The **ASME** or **ASEE** support encoding for both, constant rate, and variable rate channels. In each case the structure of the bitstream payload and the operation of the decoder are identical.

In the decoder, the output of the ASME and the output of all the available ASEE, where the previous coding layer output is available, has to be combined to form the output signal, giving the maximum audio quality. If there is an intermediate layer missing, e.g. due to transmission errors, the information in the subsequent layers can not be used. Furthermore, if there is a CELP core coder, or one or more TwinVQ layer signaled for a scalable audio object, the output of these coders has to be combined with the AAC coding layers.

A list of the valid combinations of different coding schemes is given in the subsequent subclause. The methods how to combine the individual coding layers are described in subclauses 4.5.2.2.4, 4.5.2.2.5, and 4.5.2.2.6.

4.5.2.2.3 Valid combinations of AAC with either TwinVQ or CELP

The AAC scalable coder in combination with a TwinVq or CELP base layer coder provides one way of achieving bit rate scalability. It is based on the calculation of a difference signal between the output signal of a base layer coder and the original input signal. At least one AAC extension layer is used. Joint stereo coding and mixed mono/stereo configurations are possible. All AAC joint stereo modes are available in the combined coder.

Three major classes of scalable configurations with AAC exist, depending on the coder types used:

1. AAC layers only
2. Narrow-band CELP base layer plus AAC
3. TwinVQ base layer plus AAC

In any configuration, the transmitted bandwidth (by means of max_sfb in the case of AAC and TwinVQ and by means of no_of_dc_groups or diff_short_lines in the case of CELP) of a certain layer must not be smaller than that of the preceding layer.

AAC or TwinVQ coding layer can be either coded in mono, or stereo/joint stereo. The following table summarizes the possible transitions between two layers with respect to the coder combinations and the channel configurations for each of the three coder types in mono and stereo.

Table 4.73 – Valid scalable combinations

	Layer N+1				
Layer N	NB CELP mono	TwinVQ mono	TwinVQ stereo	AAC mono	AAC stereo
Narrow Band CELP mono	X			X	X
TwinVQ mono		X		X	X
TwinVQ stereo			X		X
AAC mono				X	X
AAC stereo					X

4.5.2.2.4 Decoding of AAC only combinations

AAC only combinations basically rely on the difference encoding of the spectrum in the **S**calable **I**nverse **A**AC **Q**uantization Module (**SIAQ**) as shown in Figure 4.3 below. In the SIAQ module, after the inverse quantization, the reconstructed spectra of all individual_channel_streams are added. The number of layers is restricted to one AAC main layer and up to 7 AAC extension layers. The bitrate of the main layer and the additional layers can be any bit rate possible for AAC. However, the bitstream payload buffer restrictions given in subclause 4.5.2.7 apply. SIAQ is also a main building block for the CELP and TwinVQ combinations with AAC. (Exceptions from this are given in subclause 4.5.2.2.7)

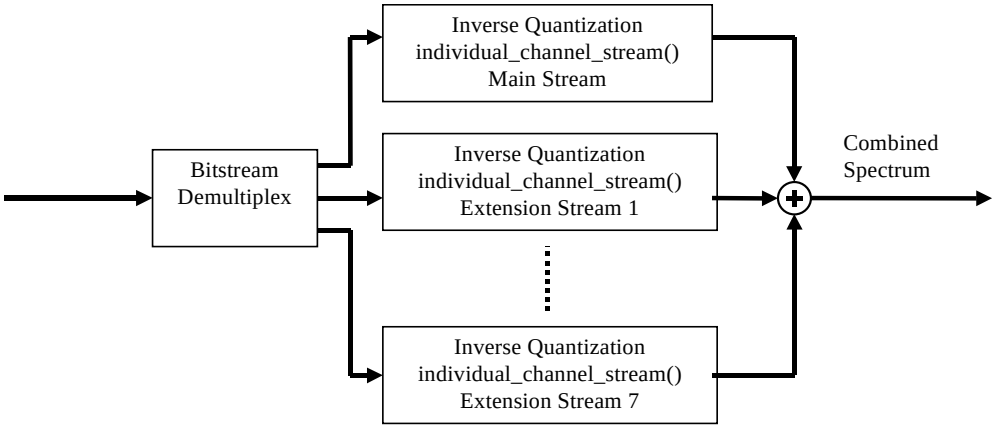


Figure 4.3 – Scalable Inverse AAC Quantization Module (SIAQ)

Three audio channel configurations of a scalable coder, based on AAC only, are possible:

AAC-Only-M:	AAC mono main layer	plus
	0 to 7 AAC mono extension layers	
AAC-Only-S:	AAC stereo main layer	plus
	0 to 7 AAC stereo extension layers	
AAC-Only-M/S:	AAC mono main layer	plus
	0 to 7 AAC mono extension layers	plus

1 to 7 AAC stereo extension layers (total number of layer <= 8)

Figure 4.4 shows the block diagram of the decoder for mode AAC-Only-M/S, which is a superset of both of the other modes. In this mode three SIAQ modules are present. The first one for the Left (L), or Mid (M), or Intensity (I) audio channel, respectively, the second for the Right (R), or Side (S), audio channel, and the third one for the mono layer (M) which, in general, is a down-mix of Left and Right which was generated in the encoder according to M = 0.5 * (L+R). For all scale factor bands where M/S coding is selected, the M-Signal is calculated by adding M'' and M' (the restrictions given in subclause 4.5.2.2.7 have to be followed which prohibit the addition under certain circumstances). For all bands where L/R coding is selected L and R are generated from L', R', and 2.0 * M'', using **diff_control_lr** to control the corresponding FSS unit. M, S, L and R are then fed into the AAC inverse joint-stereo processing module which produces the L and R spectra. After processing in the serial TNS filter combination, as described in subclauses 4.6.9 to 4.6.9.5, the time signals can be calculated via the IMDCT filterbank.

To decode mode AAC-Only-S both FSS modules and the Inverse TNS-M filters are by-passed. The M''-channel SIAQ is not present in this mode. For the decoding of mode AAC-Only-M the inverse joint stereo processing module and the Right/Side SIAQ module, as well as the corresponding TNS/IMDCT combination are not required. Only one inverse TNS filter is applied to the spectrum before the IMDCT.

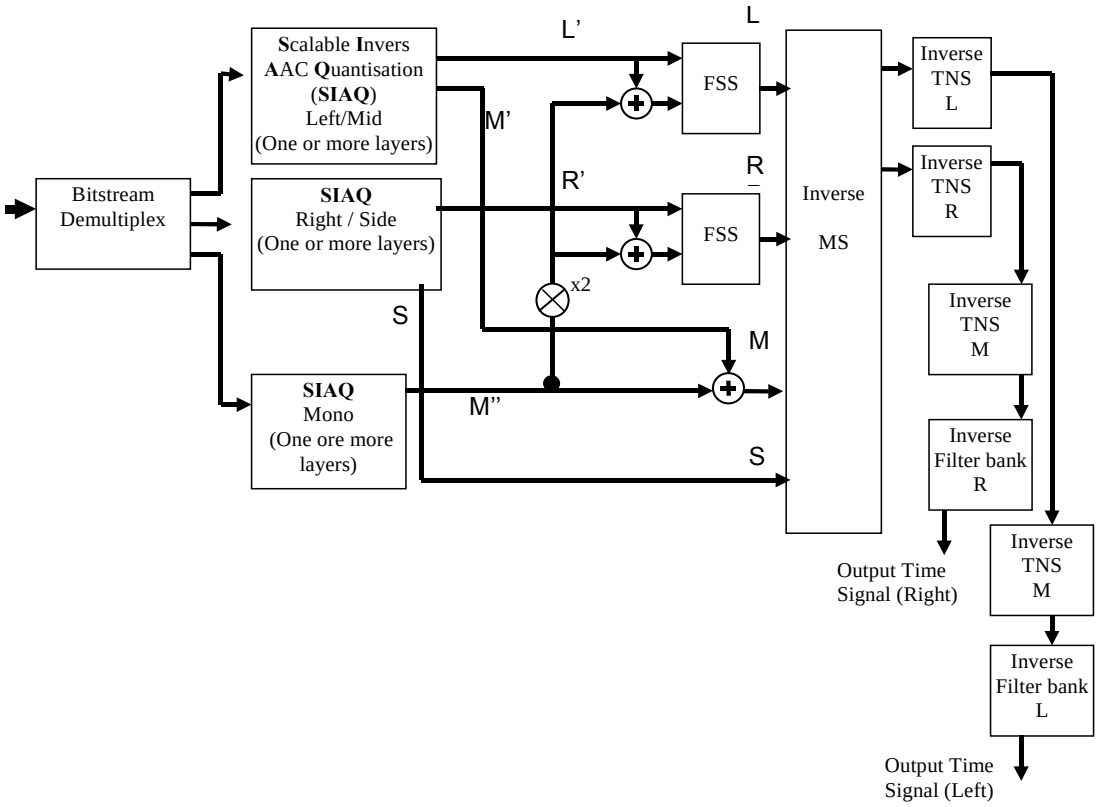


Figure 4.4 – AAC mono / AAC stereo configuration (AAC-Only-M/S)

4.5.2.2.5 Decoding of CELP / AAC combinations

This combination is primarily used to add a very low bit rate base layer in a scalable system. While in the TwinVQ/AAC combinations (see subclause 4.5.2.2.6) all coding layers work in the frequency domain, the CELP coding layer (called core coder subsequently) is a time domain coder which, in general, works with a different sampling rate. Using the up-sampling tool, which is described in subclause 4.6.15, and a FSS module (see subclause 4.6.14), the combination makes the coding gain of the CELP core coder available for the subsequent extension layers. The ratio of the sampling rate of the core coder and the sampling rate of the AAC coder always is an integer, to make the use of the upsampling tool defined in subclause 4.6.15 possible.

4.5.2.2.5.1 CELP post-filter

The post-filter of the CELP core coder has to be switched off for the output signal which is then combined with the AAC enhancement stages in the scalable coder. If only the CELP coder output is desired, the post-filter can be used to generate this output.

4.5.2.2.5.2 Frame length adaptation / super-frames

Since the AAC frame length may not necessarily be a multiple of the core frame length, the least common multiple of these two lengths can be very large. To avoid such situations an alternative AAC frame length of 960 samples, instead of 1024, is available. These frame length allows an easy integration of the MPEG-4 CELP core and AAC by constructing super-frames of a reasonable size. Some combinations of CELP-coder and GA-coder sampling rates require different numbers of core coder frames and AAC frames to build common super-frames. Table 4.74 gives an overview of the length of an AAC frame with 960 samples at various sampling rates and the length of the smallest possible super-frames for each sampling rate for all of the MPEG-4 CELP core frame length options (40, 30, 20, and 10 ms; see subpart 3, Table 3.8). Also listed in the table is the number of AAC and CELP coder frames in these smallest possible super-frames. These superframes do not exist in MPEG-4 Systems, as only single frames of each coding layer are addressed by MPEG-4 systems, making different solutions possible. However, the MPEG-4 Audio decoder is only required to support these minimum frame length superframes. Note that it is not necessary to buffer the whole super-frame. The decoder can start to produce the combined output samples whenever a sufficient number of up-sampled CELP core output samples and a full AAC layer frame is available.

In general, the CELP core signal has to be delayed after the zeo-insertion in the up-sampling tool (see subclause 4.6.15) before MDCT filterbank. The value „coreCoderDelay“ given in the GASpecificConfig() of the first AAC coding layer is the number of samples, which the core signal has to be delayed. The encoder has the choice between optimizing the total system delay for the Celp core stream (no additional delay added to the CELP coder in the encoder), or to adjust the signal of the core and AAC layer in a way so that there is no delay in the scalable decoder necessary, or any value in between those two extremes. In any case the Composition Time Stamp (if used, see ISO/IEC 14496-1) of the first Core and its corresponding first AAC frame in a superframe (see above) has to be identical. However, if only the output of the Celp core presented to the compositor and a minimum delay is desired (eg. for 2-way communication) the receiving terminal can subtract the time *td* that corresponds to **CoreCoderDelay** from the Composition Time. This is only true if there is no associated video stream for which synchronous composition has to be assured. *td* is calculated according to:

$$td = \textbf{CoreCoderDelay} / fs$$

where *fs* is the sampling rate of the AAC layer(s).

Table 4.74 – AAC frame lengths for 960 samples per frame and super-frame lengths of AAC/CELP combinations

Sampling rate (kHz)	96	64	48	32	24	16	8
AAC Frame length (ms)	10	15	20	30	40	60	120
Super-Frame length (40 ms core frame) (ms)	40	120	40	120	40	120	120
AAC / CELP frames per super-frame	4 / 1	8 / 3	2 / 1	4 / 3	1 / 1	2 / 3	1 / 3
Super-Frame length (30 ms core frame) (ms)	30	30	60	30	120	60	120
AAC / CELP frames per super-frame	3 / 1	2 / 1	3 / 2	1 / 1	3 / 4	1 / 2	1 / 4
Super-Frame length (20 ms core frame) (ms)	20	60	20	60	40	60	120
AAC / CELP frames per super-frame	2 / 1	4 / 3	1 / 1	2 / 3	1 / 2	1 / 3	1 / 6
Super-Frame length (15 ms core frame) (ms)	30	15	60	30	120	60	120
AAC / CELP frames per super-frame	3 / 2	1 / 1	3 / 4	1 / 2	3 / 8	1 / 4	1 / 8
Super-Frame length (10 ms core frame) (ms)	10	30	20	30	40	60	120
AAC / CELP frames per super-frame	1 / 1	2 / 3	1 / 2	1 / 3	1 / 4	1 / 6	1 / 12

4.5.2.2.5.3 CELP core coder with AAC running at 88.2 kHz, 44.1 kHz, or 22.05 kHz sampling rate

AAC frames using sampling rates of 88.2 kHz, 44.1 kHz, or 22.05 kHz can be achieved by adjusting the sampling rate of the CELP core coder such, that an integer ratio between these two sampling rate is achieved. Table 4.112 shows the mapping of the AAC sampling rates to the CELP core coder sampling rates. The CELP core coder runs with the sampling rate listed in this table. The CELP decoding process is completely identical to the methods defined for a sampling rate of 8 kHz for the narrow band CELP coder. Table 4.75 shows the super frame parameters for the AAC sampling rates 88.2 kHz, 44.1 kHz, and 22.05 kHz.

Table 4.75 – Super-frame parameters of AAC/CELP combinations at AAC sampling rates of 88.2 kHz, 44.1 kHz and 22.05 kHz

Sampling rate AAC (kHz)	88.2	44.1	22.05
AAC Frame length (ms)	10.884	21.768	43.537
Super-frame length (43.537 ms core frame) (ms)	43.537	43.537	43.537
AAC / CELP frames per super-frame	4/1	2 / 1	1 / 1
Super-frame length (32.653 ms core frame) (ms)	32.653	65.306	130.612
AAC / CELP frames per super-frame	3/1	3 / 2	3 / 4
Super-frame length (21.768 ms core frame) (ms)	21.768	21.768	43.537
AAC / CELP frames per super-frame	2/1	1 / 1	1 / 2
Super-frame length (16.326 ms core frame) (ms)	32.653	65.306	130.612
AAC / CELP frames per super-frame	3 / 2	3 / 4	3 / 8
Super-frame length (10.884 ms core frame) (ms)	10.884	21.768	43.537
AAC / CELP frames per super-frame	1/1	1 / 2	1 / 4

The possible sampling rates for the CELP core in this case are 7350 Hz (narrow-band core) or 14700 Hz (wide-band core).

4.5.2.2.5.4 CELP / AAC Configurations

Several CELP/AAC combinations, which differ in the number of audio channels in each layer, are defined:

CELP-AAC-M	CELP mono layer	plus
	AAC mono main layer	plus
	0 to 7 AAC mono extension layer	
CELP-AAC-MS	CELP mono layer	plus
	AAC mono main layer	plus
	0 to 7 AAC extension layer	plus
	0 to 7 AAC stereo extension layer	(total number of AAC layer <= 8)
CELP-AAC-S	CELP mono layer	plus
	AAC stereo main layer	plus
	0 to 7 AAC stereo extension layer	

Note that the CELP coder also has an intrinsic scalability function (see subpart 3). This means that the CELP coding layer itself might consist of several narrow-band CELP coding layers.

Figure 4.5 below shows the decoder block diagram of the CELP-AAC-M configuration.:

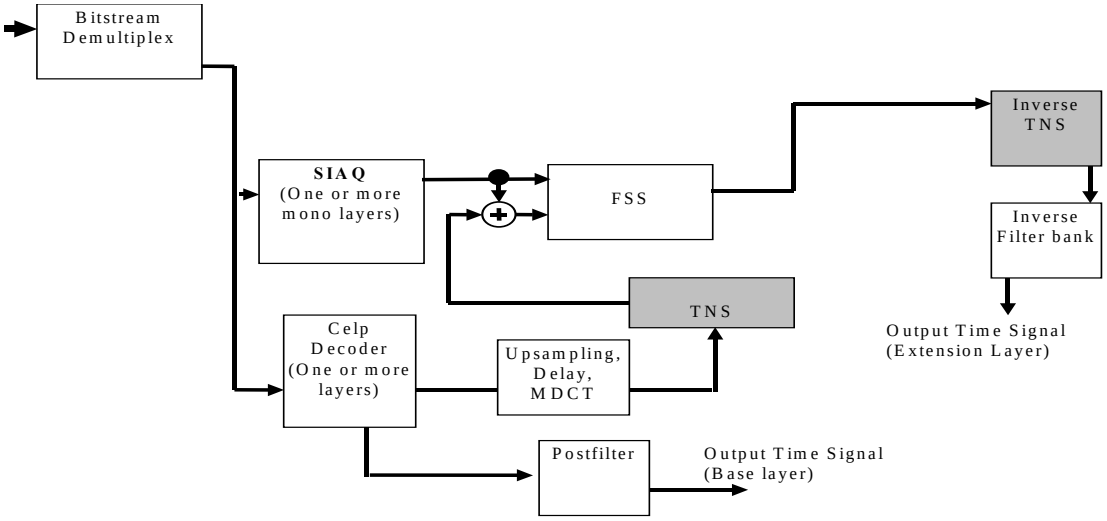


Figure 4.5 – Block diagram of the Celp + AAC mono combination

The CELP core is decoded and the output is up-sampled, delayed and transformed into the frequency domain as described in subclause 4.6.15. This signal is then combined with the reconstructed AAC layer spectrum in the Frequency Selective Switch (FSS, see subclause 4.6.14) tool to get the complete output spectrum. If TNS is used the encoder TNS filter is applied to the spectrum of the core coder. The normal decoder TNS-filter is then applied before the calculation of the inverse IMDCT Filterbank as usual.

Figure 4.6 shows mode CELP-AAC-S, where a mono CELP core is combined with a stereo AAC layer. The TNS filter is applied to the MDCT coefficients calculated from the upsampled CELP decoder output. The inverse TNS filter is needed only for the enhancement layer output.

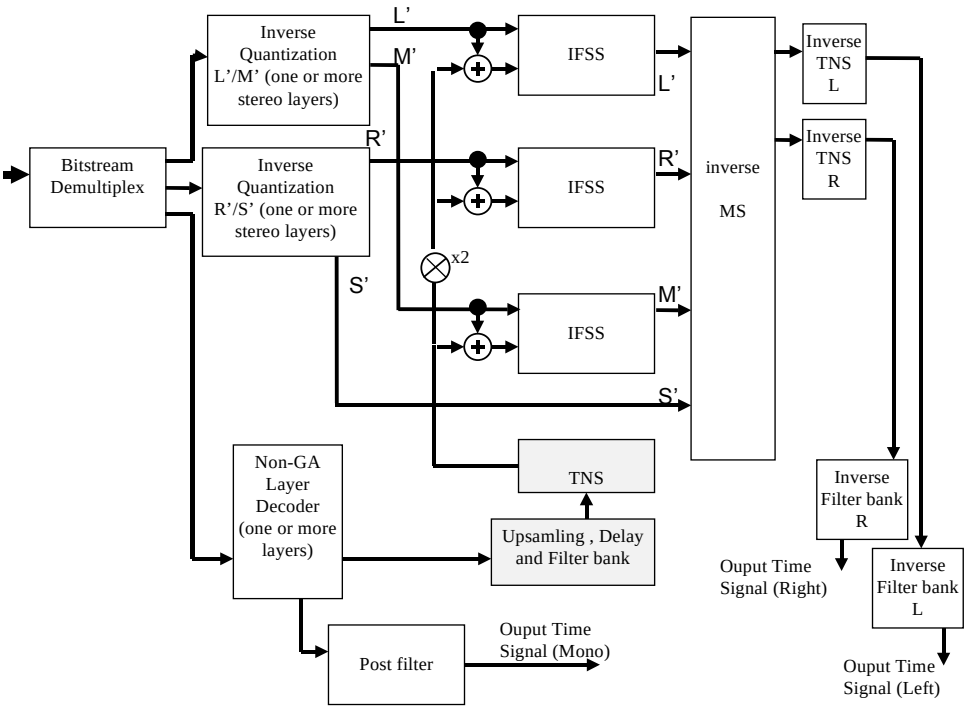


Figure 4.6 – Decoding of a non-GA (i.e. CELP) mono plus AAC stereo combination

Figure 4.7 shows mode CELP-AAC-MS, where a non-GA (i.e. CELP) mono layer, and at least one AAC mono layer, plus at least one AAC stereo layer is combined. The TNS filter for the Mid signal is applied to the MDCT coefficients reconstructed from the CELP core. Both inverse TNS filter, the inverse filter for the M signal, and the inverse TNS filter for the L/R signal are applied to the final layer output as described in subclause 4.6.9 to 4.6.9.5.

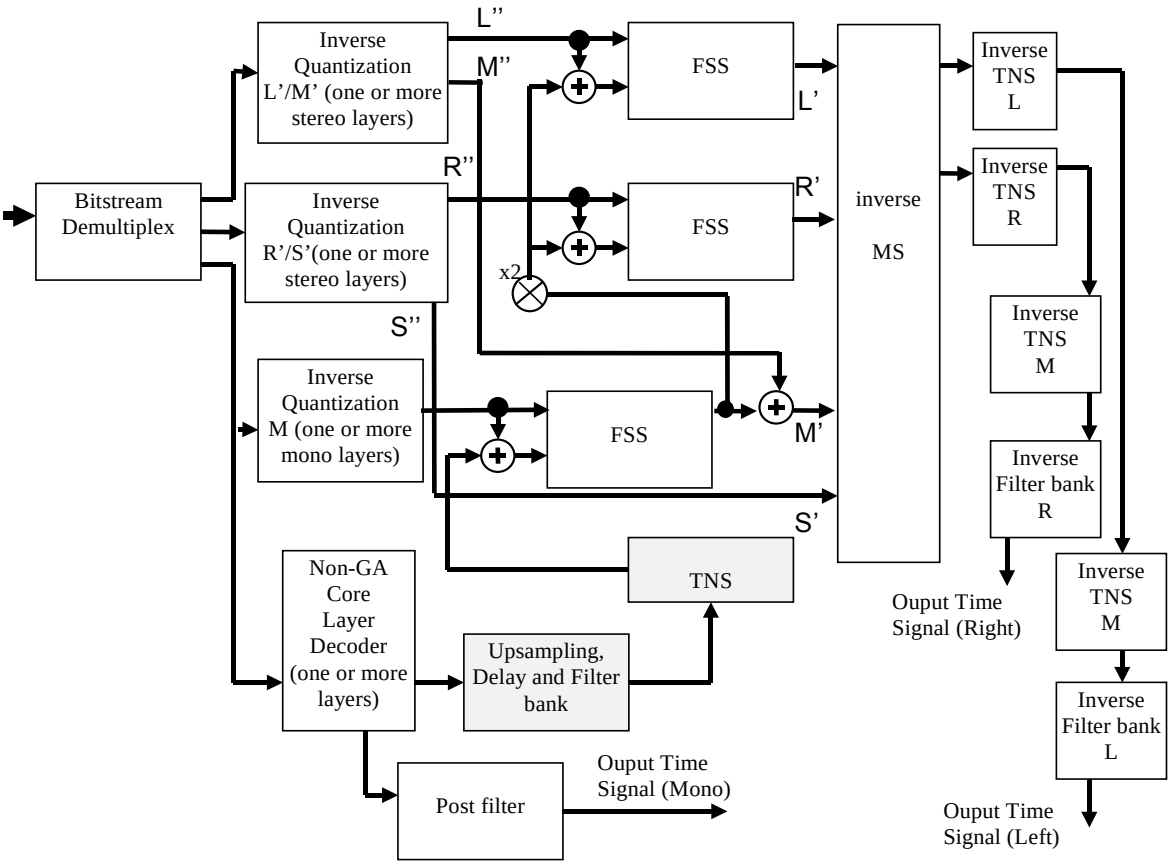


Figure 4.7 – Decoding of a Celp(mono) + AAC(mono) + AAC(stereo) configuration

4.5.2.2.5.5 Alternative core coder

Within MPEG-4, only the MPEG-4 CELP coder in a single layer configuration is available as core coder. However, there is no principal limitation as to which core coder can be used, although, in general, the core coder should encode the waveform of the input signal, to allow a useful difference signal to be calculated. Especially CELP coder with a frame length of a multiple of 10 ms can easily be integrated.

4.5.2.2.6 Decoding of TwinVQ / AAC combinations

It is possible to construct a scalable coder by combining TwinVQ and AAC. This configuration provides the advantages of both coders: TwinVQ can reduce the averaged distortion at the low bit rate range and AAC is good at controlling the quantization noise for high quality coding. The scalable bitstream payload is first demultiplexed and the spectral information of the TwinVQ and the AAC layer are extracted, as well as the side information. After the AAC inverse quantization, the addition of the two spectra is performed considering the FSS control information. Finally, time domain signal is generated through IMDCT.

One advantage of the combined TwinVQ - AAC audio coding is that both quantization schemes operate on the same signal space, i.e. MDCT coefficients. Also, both coders share the same sampling rate. Therefore no ”interfacing” is necessary to convert between the internal representations of the audio data of the two coders, like for the CELP coder, where an additional up-sampling filterbank has to be calculated.

4.5.2.2.6.1 Combination with the AAC tools

The AAC oriented tools such as TNS, LTP and joint stereo coding can be combined with TwinVQ and the scalable coder. LTP is applied only to the base layer and other tools are commonly used for both TwinVQ layer and AAC layer. There are number of possibilities how to combine these tools. The combinations of TwinVQ with TNS are shown in Figure 4.8 to Figure 4.15. The transmission of tns_data_present and tns_data() in the first AAC layer depends on the actually used configuration. Table 4.76 shows the conditions when the TNS related data elements are transmitted, and when they are not.

Table 4.76 – Values of helper element tns_aac_tvq_en[] depending on tvq_mono_tns, tvq_stereo_tns_left and tvq_stereo_tns_right.

	tns_aac_tvq_en[0]	tns_aac_tvq_en[1]
(tvq_mono_tns==1) && (this_layer_stereo == 1)	1	1
(tvq_mono_tns==1) && (this_layer_stereo == 0)	0	X
(tvq_mono_tns==0)	1	1
(tvq_stereo_tns_left==1) && (tvq_stereo_tns_right==1)	0	0
(tvq_stereo_tns_left==1) && (tvq_stereo_tns_right==0)	0	1
(tvq_stereo_tns_left==0) && (tvq_stereo_tns_right==1)	1	0
(tvq_stereo_tns_left==0) && (tvq_stereo_tns_right==0)	1	1

4.5.2.2.6.2 max_sfb and scale_factor_grouping for TwinVQ

The scalefactor band tables and the parameter **max_sfb** are necessary for most of the AAC related tools, such as LTP, TNS, FSS (diff_control) and joint stereo coding. If TwinVQ is combined with these tools, the scalefactor bands as defined in Table 4.110 to Table 4.128 (see subclause 4.5.4) are used. On the other hand, TwinVQ normally does not use the parameter **max_sfb.** However, it can be calculated from the frequency range that is used in the interleaved vector quantization as shown in the following pseudo-code:

```
tvq_main_element():
        max_line = (int) (frame_length/num_windows * qsample);
        for (iw = 0, i_sfb = 0; ((iw < num_swb) && (swb_offset[iw] < max_line)); iw++){
                i_sfb = iw+1;
        }
        max_sfb = i_sfb;


tvq_extension_element():
        max_line = (int) (frame_length /num_windows* (qsample+bias*3));
        for (iw = 0, i_sfb = 0; ((iw < num_swb) && (swb_offset[iw] < max_line)); iw++) {
                i_sfb = iw+1;
        }
        max_sfb[lyr] = i_sfb;
```

The parameter, max_line is a highest frequency line that the interleave VQ quantizes. Frame_length is either 1024 or 960. In case of EIGHT_SHORT_SEQUENCES, num_windows is 8, otherwise it is 1. The values of **qsample** and **bias** are defined in the section of adaptive active band selection (subclause 4.6.5.4), and depend only on the sampling frequency and the bitrate. The tables of swb_offset are listed in subclause 4.5.4. Note that the parameter of **scale_factor_grouping** is transmitted in the TwinVQ element if it is needed for stereo joint coding with TwinVQ.

4.5.2.2.6.3 Audio channel configurations

TVQ-AAC-MM	TwinVQ mono layer	plus
	AAC mono main layer	plus
	0 to 7 AAC mono extension layer	
TVQ-AAC-MMS	TwinVQ mono layer	plus
	AAC mono main layer	plus
	0 to 7 AAC mono extension layer	plus

	0 to 7 AAC stereo extension layer	(total number of AAC layer <= 8)
TVQ-AAC-MS	TwinVQ mono layer	plus
	AAC stereo main layer	plus
	0 to 7 AAC stereo extension layer	
TVQ-AAC-SS	TwinVQ stereo layer	plus
	AAC stereo main layer	plus
	0 to 7 AAC stereo extension layer	

Figure 4.8 shows the case where TwinVQ is mono, and TwinVQ does not use TNS, where the TNS filter is applied to the reconstructed MDCT coefficients of the TwinVQ decoder. The inverse TNS filters for the L- and R-channel are needed only for the extension layer output.

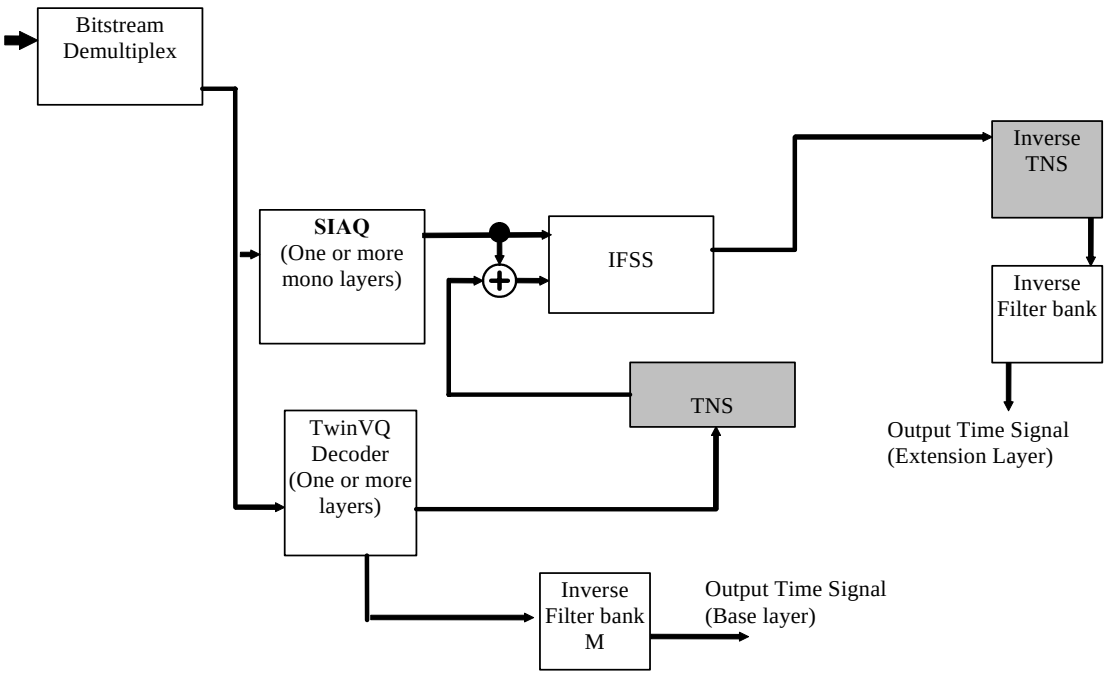


Figure 4.8 – Decoding of a TwinVQ (mono) + AAC (mono) configuration, TwinVQ does not use TNS

Figure 4.9 shows the case where TwinVQ is mono, and uses TNS, where the inverse TNS filter is applied for both, for the TwinVQ only output, and the combined TwinVQ-AAC output signal.

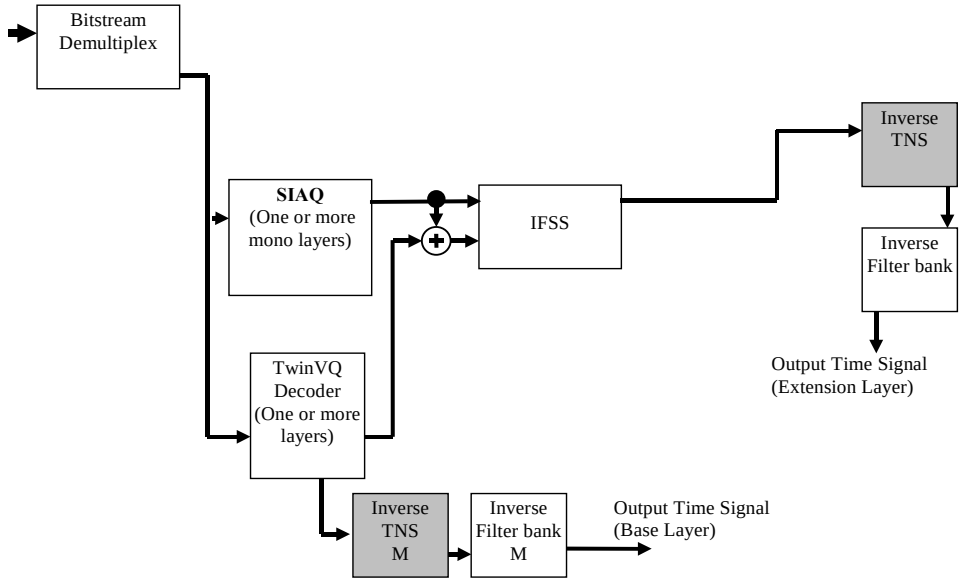


Figure 4.9 – Decoding of a TwinVQ (mono) + AAC (mono) configuration, TwinVQ does use TNS

Figure 4.10 shows the case where TwinVQ is mono, and uses TNS, and at least one AAC mono and one AAC stereo layer is present, where the both the middle, and the left/right inverse TNS filter is applied to the left and right spectrum before calculating the IMDCT.

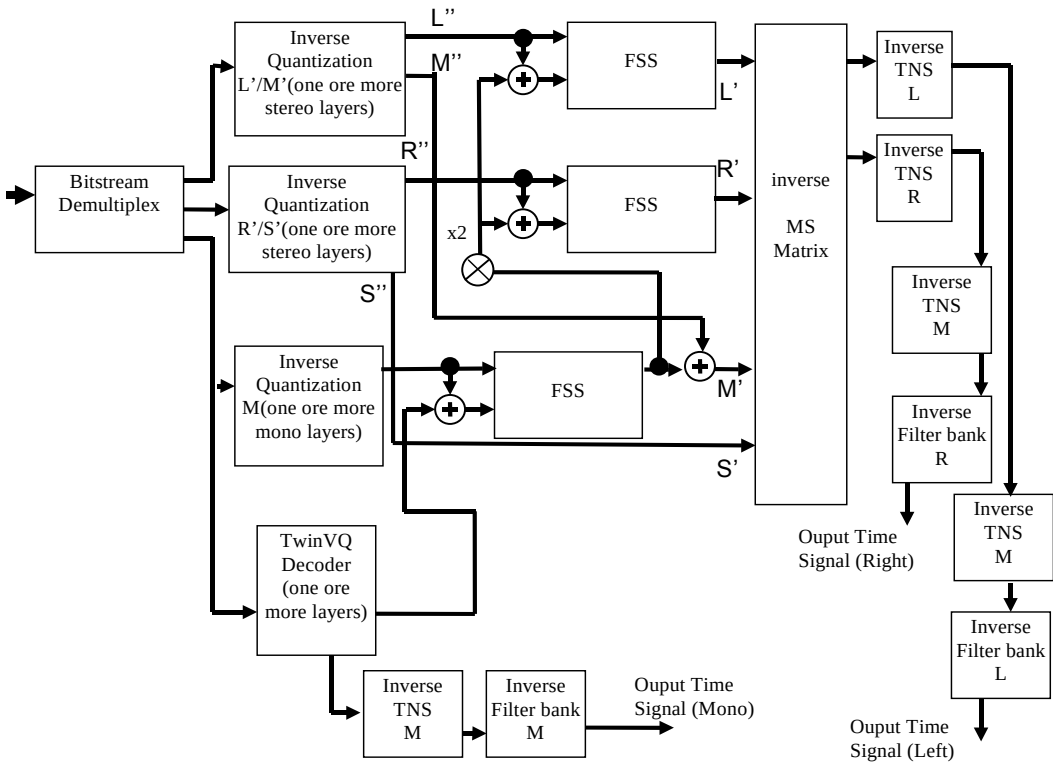


Figure 4.10 – Decoding of a TwinVQ (mono) AAC (mono) and AAC (stereo) configuration, TwinVQ does use TNS

Figure 4.11 shows the case where TwinVQ is mono and does **not** use TNS, and at least one AAC mono and one AAC stereo layer is present, where both, the middle, and the left/right chanel inverse TNS filter is applied to the left and right spectrum before calculating the IMDCT.

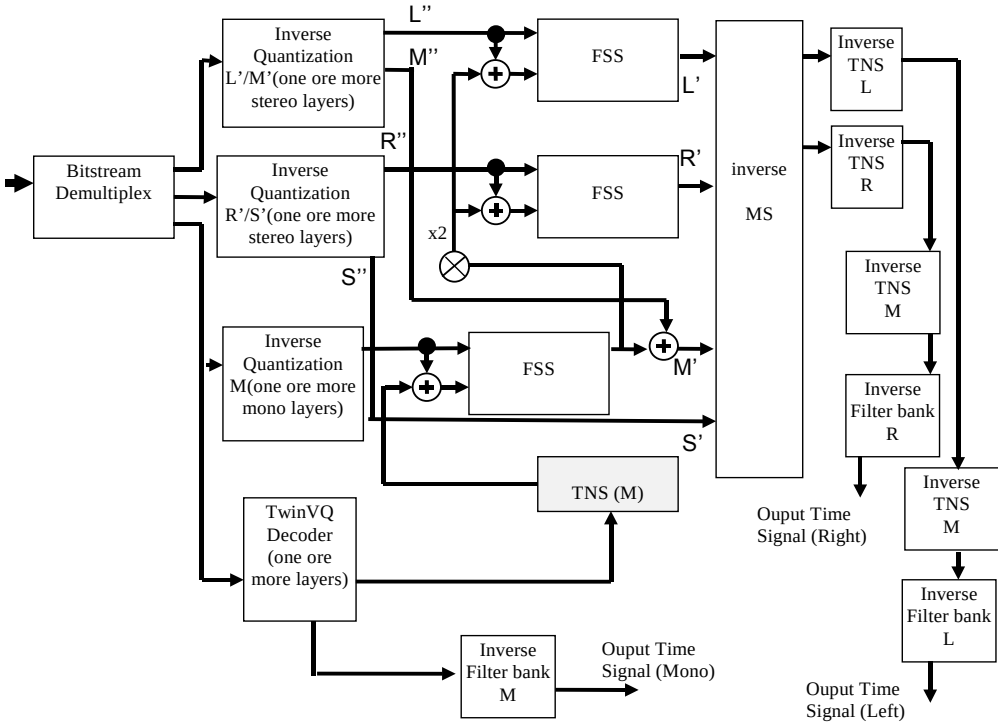


Figure 4.11 – Decoding of a TwinVQ (mono) + AAC (mono) + AAC (stereo) configuration, if TNS is NOT used in the TwinVQ layer

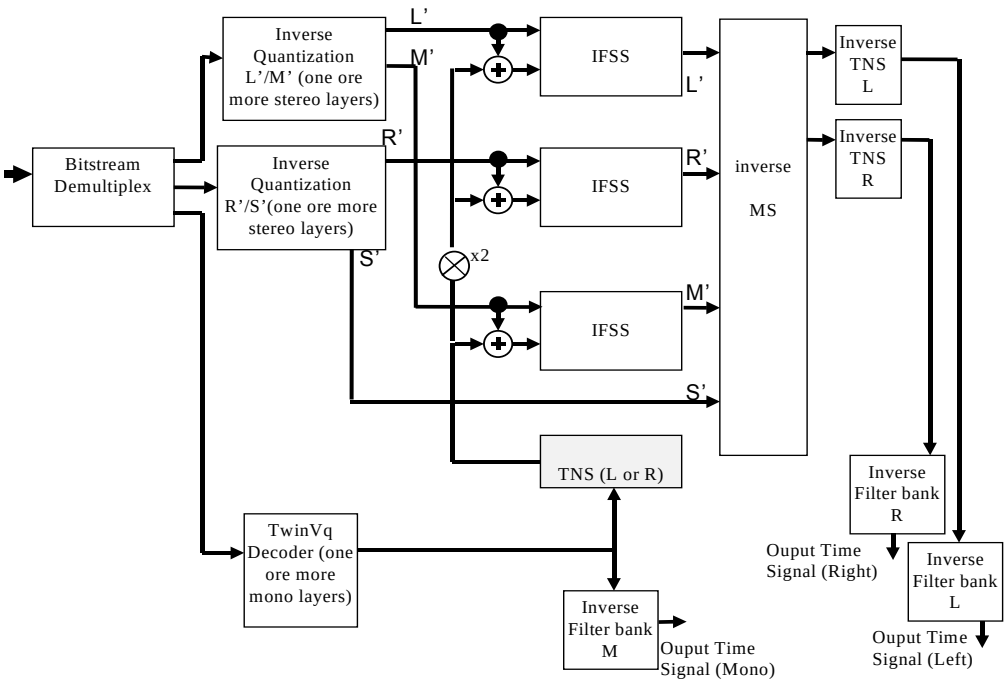


Figure 4.12 – Decoding of a TwinVQ (mono) + AAC (stereo) configuration, if TNS is NOT used in the TwinVQ layer

Figure 4.13 shows the case that TwinVQ is stereo and TwinVQ does not use TNS. Here a TNS filter is applied to the reconstructed MDCT coefficients of the TwinVQ decoder. The inverse TNS filter is needed only for the enhancement layer output.

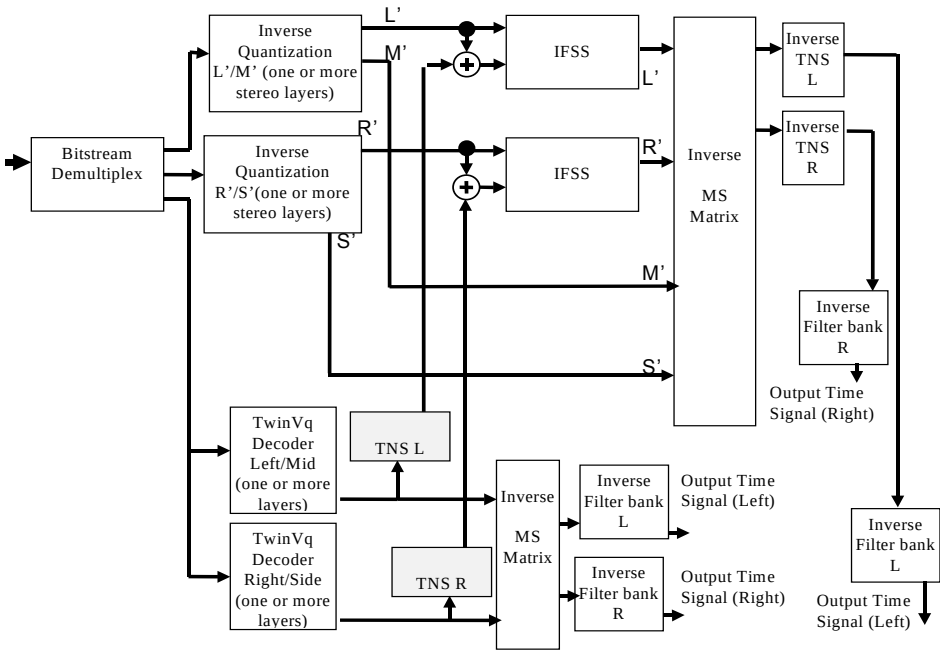


Figure 4.13 – Decoding of a TwinVQ (stereo) +AAC (stereo) configuration, if TNS is NOT used in the TwinVQ layer

Figure 4.14 shows the case where TwinVQ is stereo, and TwinVQ uses TNS, where the common inverse TNS filter is applied to both, the base layer, and the enhancement layer output.

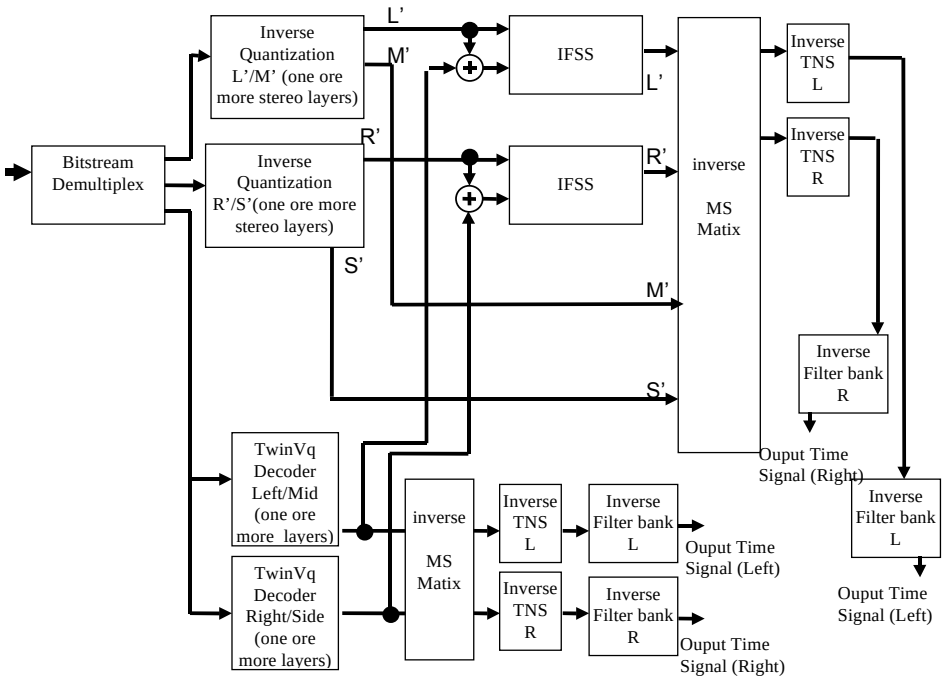


Figure 4.14 – Decoding of a TwinVQ (stereo) + AAC (stereo) configuration, if TNS is used in a TwinVQ layer

Figure 4.15 shows the case where TwinVQ is mono, and TwinVQ uses TNS, where the common inverse TNS filter is applied to both, the base layer, and the enhancement layer output.

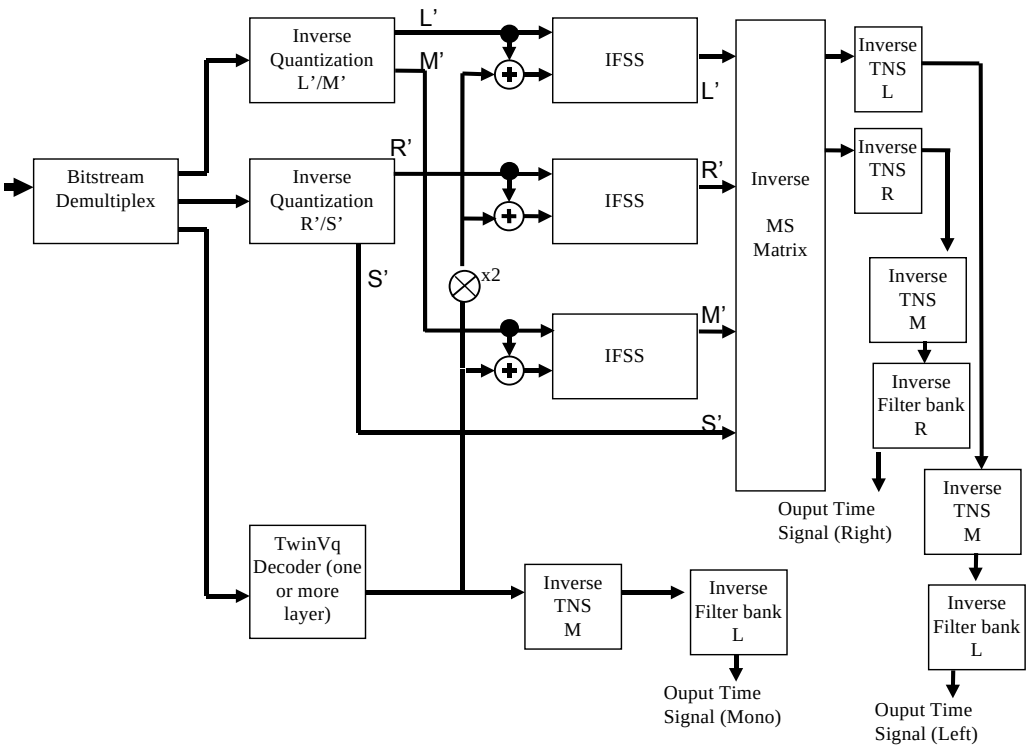


Figure 4.15 – Decoding of a TwinVQ (mono) + AAC (stereo) configuration, if TNS is used in the first TwinVQ layer

4.5.2.2.7 Combining AAC layers if PNS, MS or Intensity tools are used in a particular scale factor band

The following tables specify the output spectrum of a particular scale factor band of the combined layers N and N+1 for various combinations of the PNS, Intensity, and MS coding tools in layer N and layer N+1 for different layer combinations:

Table 4.77 – Mono-mono layer combination

Tool used in Layer N	Tool used in Layer N+1	Output of the combined Layer:
No Tool	No Tool	Sum of Layer N and Layer N+1
No Tool	PNS	Invalid combination
PNS	No Tool	see subclause 4.6.13.6
PNS	PNS	Layer N+1

Table 4.78 – Stereo-stereo layer combination

Tool used in Layer N	Tool used in Layer N+1	Output of the combined Layer:
No Tool or MS	No Tool or MS	Sum of Layer N and Layer N+1
No Tool or MS	PNS	Invalid combination
No Tool or MS	Intensity	Invalid combination
No Tool or MS	PNS & Intensity	Invalid combination
PNS	No Tool	see subclause 4.6.13.6
PNS	MS	see subclause 4.6.13.6
PNS	Intensity	Layer N+1
PNS	PNS	Layer N+1
PNS	PNS & Intensity	Layer N+1
Intensity	No Tool or MS	Layer N+1
Intensity	PNS	Invalid combination
Intensity	Intensity	Sum of Layer N and Layer N+1, only M/L-channel; Take Positions from Layer N+1
Intensity	PNS & Intensity	Invalid combination
PNS & Intensity	No Tool or MS	Layer N+1
PNS & Intensity	PNS	Invalid combination
PNS & Intensity	Intensity	Layer N+1
PNS & Intensity	PNS & Intensity	Layer N+1

Table 4.79 – Mono-stereo layer combination

Tool used in Layer N	Tool used in Layer N+1	Output of the combined Layer:
No Tool	No Tool	Sum of Layer N and Layer N+1 (FSS-Tool)
No Tool	MS	Sum of Layer N and Layer N+1
No Tool	PNS	Invalid combination
No Tool	Intensity	Layer N+1
No Tool	PNS & Intensity	Layer N+1
PNS	No Tool	Layer N+1
PNS	MS	Layer N+1
PNS	Intensity	Layer N+1
PNS	PNS	Layer N+1
PNS	PNS & Intensity	Layer N+1

4.5.2.3 Decoding of an individual_channel_stream (ICS) and ics_info

The Individual Channel Stream (ICS) elements are sub-elements of both, the MPEG-2 AAC style audio object types (AAC Main, LC, SSR, LTP), and the scalable AAC audio object type (AAC Scalable).

4.5.2.3.1 Definitions

4.5.2.3.1.1 Data elements

individual_channel_stream()	contains data necessary to decode one channel
ics_info()	contains side information necessary to decode an individual_channel_stream for SCE and CPE elements. The individual_channel_streams of a channel_pair_element() may share one common ics_info
ics_reserved_bit	flag reserved for future use. Shall be '0'.
window_sequence	indicates the sequence of windows as defined in Table 4.109.
window_shape	A 1 bit field that determines what window is used for the right hand part of this analysis window

max_sfb	number of scalefactor bands transmitted per group either in ics_info(), aac_scalable_main_element(), or aac_scalable_extension_element(). For TwinVQ elements max_sfb is not transmitted directly, but is calculated in the decoder from other transmitted information as specified in subclause 4.5.2.2.6.1.
scale_factor_grouping	a bit field that contains information about grouping of short spectral data

4.5.2.3.1.2 Help elements

scalefactor window band	term for scalefactor bands within a window, given in Table 4.110 to Table 4.128.
scalefactor band	term for scalefactor band within a group. In case of EIGHT_SHORT_SEQUENCE and grouping a scalefactor band may contain several scalefactor window bands of corresponding frequency. For all other window_sequences scalefactor bands and scalefactor window bands are identical.
g	group index
win	window index within group
sfb	scalefactor band index within group
swb	scalefactor window band index within window
bin	coefficient index
num_window_groups	number of groups of windows which share one set of scalefactors
window_group_length[g]	number of windows in each group.
bit_set(bit_field,bit_num)	function that returns the value of bit number bit_num of a bit_field (most right bit is bit 0)
num_windows	number of windows of the actual window sequence
num_swb_long_window	number of scalefactor bands for long windows. This number has to be selected depending on the sampling frequency. See subclause 4.5.4.
num_swb_short_window	number of scalefactor window bands for short windows. This number has to be selected depending on the sampling frequency. See subclause 4.5.4.
num_swb	number of scalefactor window bands for short windows in case of EIGHT_SHORT_SEQUENCE, number of scalefactor window bands for long windows otherwise
swb_offset_long_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for long windows. This table has to be selected depending on the sampling frequency.
swb_offset_short_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows. This table has to be selected depending on the sampling frequency.
swb_offset[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows
sect_sfb_offset[g][section]	table that gives the number of the start coefficient for the section_data() within a group. This offset depends on the window_sequence and scale_factor_grouping.
sampling_frequency_index	see subclause 1.6.3.3 of subpart 1.

4.5.2.3.2 Decoding process

Decoding an individual_channel_stream (ICS)

In the individual_channel_stream, the order of decoding is:

Get global_gain

Get ics_info (parse bitstream payload if common information is not present)

Get section_data, if present

Get scale_factor_data, if present

Get pulse_data if present

Get tns_data, if present

Get gain_control_data, if present

If the HCR tool is not used:

Get spectral_data, if present

If the HCR tool is used:

Get length_of_reordered_spectral_data, if present

Get length_of_longest_codeword, if present

Get reordered_spectral_data, if present

The process of recovering pulse_data is described in subclause 4.6.3, tns_data in subclause 4.6.9, and gain_control data in subclause 4.6.12. An overview of how to decode ics_info section_data, scalefactor_data, and spectral_data will be in the following paragraphs.

Recovering ics_info()

For single_channel_element()'s ics_info is always located immediately after the global_gain in the inidividual_channel_stream. For a channel pair element there are two possible locations for the ics_info.

In case of channel_pair_element() if the common_window flag is set to 1 both channels share the same ics_info() (i.e. both have same window_sequence, same window_shape, same scale_factor_grouping, same max_sfb etc.). Otherwise (i.e. common_window is set to 0) there is an ics_info immediately after the global_gain for each of the two individual_channel_stream().

The ics_info() carries the window information associated with an ICS and thus permits channels in a channel_pair to switch separately if desired. In addition it carries the max_sfb which places an upper limit on the number of ms_used[] and predictor_used[] bits that must be transmitted. If the window_sequence is EIGHT_SHORT_SEQUENCE then scale_factor_grouping is transmitted. If a set of short windows form a group then they share scalefactors as well as intensity stereo positions and PNS information and have their spectral coefficients interleaved. The first short window is always a new group so no grouping bit is transmitted. Subsequent short windows are in the same group if the associated grouping bit is 1. A new group is started if the associated grouping bit is 0. It is assumed that grouped short windows have similar signal statistics. Hence their spectra are interleaved to place correlated coefficients next to each other. The manner of interleaving is indicated in Figure 4.24. The ics_info() also carries the prediction_data() for the individual channel or channel pair (see subclause 4.6.6 and 4.6.7).

Recovering section_data()

In the ICS, the information about one long window, or eight short windows, is recovered. The section_data() is the first field to be decoded, and describes the Huffman codes that apply to the scalefactor bands in the ICS. The form of the section data is:

sect_cb The codebook for the section

and

sect_len The length of the section.

This length is recovered by reading the bitstream payload sequentially for a section length, adding the escape value to the total length of the section until a non-escape value is found, which is added to establish the total length of the section. This process is clearly explained in the C-like syntax description. If the aacSectionDataResilienceFlag is set, **sect_len_incr** is not transmitted but is set to one per default in case the codebook for a section is 11 or in the range of 16 and 31. Note that within each group the sections must delineate the scalefactor bands from zero to **max_sfb** so that the first section within each group starts at bands zero and the last section within each group ends at **max_sfb**.

The sectioning data describes the codebook, and then the length of the section using that codebook, starting from the first scalefactor band and continuing until the total number of scalefactor bands is reached.

After this description is provided, all scalefactors and spectral data corresponding to codebook zero are zeroed, and no values corresponding to these scalefactors or spectral data will be transmitted. When scanning for scalefactor data it is important to note that scalefactors for any scalefactor bands whose Huffman codebook is zero will be omitted. Similarly, all spectral data associated with Huffman codebook zero are omitted.

In addition spectral data associated with the scalefactor bands that have an intensity codebook will not be transmitted, but intensity stereo positions will be transmitted in place of the scalefactors, as described in section 4.6.8.1.4.

scalefactor_data() parsing and decoding

For each scalefactor band that is not in a section coded with the zero codebook (ZERO_HCB), a scalefactor is transmitted. These will be denoted as 'active' scalefactor bands and the associated scalefactors as active scalefactors. Global gain, the first data element in an ICS, is typically the value of the first active scalefactor. All scalefactors (and also the stereo positions and pns energies) are transmitted using Huffman coded DPCM relative to the previous active scalefactor (respectively previous stereo position or previous pns energy, see subclause 4.6.2 and 4.6.3). The first active scalefactor is differentially coded relative to the global gain. Note that it is not illegal, merely inefficient, to provide a global_gain that is different from the first active scalefactor and then a non-zero DPCM value for the first scalefactor DPCM value. If any intensity stereo positions are received interspersed with the DPCM scalefactor elements, they are sent to the intensity stereo module, and are not involved in the DPCM coding of scalefactor values. This also applies for PNS energies. The value of the first active scalefactor is usually transmitted as the global_gain with the first DPCM scalefactor having a zero value. Once the scalefactors are decoded, the actual values are found via a power function.

spectral_data() parsing and decoding

The spectral data is recovered as the last part of the parsing of an ICS. It consists of all the non-zeroed coefficients remaining in the spectrum or spectra, ordered as described in the ICS_info. For each non-zero, non-intensity codebook, the data are recovered via Huffman decoding in quads or pairs, as indicated in the noiseless coding tool (see subclause 4.6.9). If the spectral data is associated with an unsigned Huffman codebook, the necessary sign bits follow the Huffman codeword. In the case of the ESCAPE codebook, if any escape value is received, a corresponding escape sequence will appear after that Huffman code. There may be zero, one or two escape sequences for each codeword in the ESCAPE codebook, as indicated by the presence of escape values in that decoded codeword. For each section the Huffman decoding continues until all the spectral values in that section have been decoded. Once all sections have been decoded, the data is multiplied by the decoded scalefactors and deinterleaved if necessary.

If the HCR tool is used, spectral data does not consist of consecutive codewords anymore. Concerning HCR, the whole data necessary to decode two or four lines are referred as codeword. This includes Huffman codeword, sign bits, and escape sequences.

4.5.2.3.3 Windows and window sequences

Quantization and coding is done in the frequency domain. For this purpose, the time signal is mapped into the frequency domain in the encoder. The decoder performs the inverse mapping as described in subclause 4.6.11. Depending on the signal, the coder may change the time/frequency resolution by using two different windows: LONG_WINDOW and SHORT_WINDOW. To switch between windows, the transition windows LONG_START_WINDOW and LONG_STOP_WINDOW are used. Table 4.108 lists the windows, specifies the corresponding transform length and shows the shape of the windows schematically. Two transform lengths are used: 1024 (or 960) (referred to as long transform) and 128 (or 120) coefficients (refered to as short transform).

Window sequences are composed of windows in a way that a raw_data_block always contains data representing 1024 (or 960) output samples. The data element **window_sequence** indicates the window sequence that is

actually used. lists how the window sequences are composed of individual windows. Refer to subclause 4.6.11 for more detailed information about the transform and the windows.

4.5.2.3.4 Scalefactor bands and grouping

Many tools of the decoder perform operations on groups of consecutive spectral values called scalefactor bands (abbreviation 'sfb'). The width of the scalefactor bands is built in imitation of the critical bands of the human auditory system. For that reason the number of scalefactor bands in a spectrum and their width depend on the transform length and the sampling frequency.Table 4.110 to Table 4.128 list the offset to the beginning of each scalefactor band on the transform lengths 1024 (960) and 128 (120) and on the sampling frequencies.

To reduce the amount of side information in case of sequences which contain SHORT_WINDOWS, consecutive SHORT_WINDOWs may be grouped (see Figure 4.22). The information about the grouping is contained in the **scale_factor_grouping** data element. Grouping means that only one set of scalefactors is transmitted for all grouped windows as if there was only one window. The scalefactors are then applied to the corresponding spectral data in all grouped windows. To increase the efficiency of the noiseless coding (see subclause 4.6.3), the spectral data of a group is transmitted in an interleaved order given in subclause 4.5.2.3.5. The interleaving is done on a scalefactor band by scalefactor band basis, so that the spectral data can be grouped to form a virtual scalefactor band to which the common scalefactor can be applied. Within this document the expression 'scalefactor band' (abbreviation 'sfb') denotes these virtual scalefactor bands. If the scalefactor bands of the single windows are referred to, the expression 'scalefactor window band' (abbreviation 'swb') is used. Due to its influence on the scalefactor bands, grouping affects the meaning of section_data (see subclause 4.6.3), the order of spectral data (see subclause 4.5.2.3.5), and the total number of scalefactor bands. For a LONG_WINDOW scalefactor bands and scalefactor window bands are identical since there is only one group with only one window.

To reduce the amount of information needed for the transmission of side information specific to each scalefactor band, the data element **max_sfb** is transmitted. Its value is one greater than the highest active scalefactor band in all groups. **max_sfb** has influence on the interpretation of section data (see subclause 4.6.3), the transmission of scalefactors (see subclause 4.6.3 and 4.6.2), the transmission of predictor data (see subclause 4.6.6 and 4.6.7), the FSS control information in the mono-stereo coding modes (see subclause 4.6.14 and 4.5.2.2), and the transmission of the ms_mask (see subclause 4.6.8.1).

Since scalefactor bands are a basic element of the coding algorithm, some help variables and arrays are needed to describe the decoding process in all tools using scalefactor bands. These help variables depend on sampling_frequency, **window_sequence, scalefactor_grouping** and **max_sfb** and must be built up for each raw_data_block. The pseudo code shown below describes

- how to determine the number of windows in a window_sequence named *num_windows*
- how to determine the number of window_groups named *num_window_groups*
- how to determine the number of windows in each group named *window_group_length[g]*
- how to determine the total number of scalefactor window bands named *num_swb* for the actual window type
- how to determine *swb_offset[swb]*, the offset of the first coefficient in scalefactor window band named *swb* of the window actually used
- how to determine *sect_sfb_offset[g][section],* the offset of the first coefficient in section named *section*. This offset depends on **window_sequence** and **scale_factor_grouping** and is needed to decode the spectral_data().

A long transform window is always described as a window_group containing a single window. Since the number of scalefactor bands and their width depend on the sampling frequency, the affected variables are indexed with sampling_frequency_index to select the appropriate table.

```
fs_index = sampling_frequency_index;
switch (window_sequence) {
   case ONLY_LONG_SEQUENCE:
   case LONG_START_SEQUENCE:
   case LONG_STOP_SEQUENCE:
      num_windows = 1;
      num_window_groups = 1;
      window_group_length[num_window_groups-1] = 1;
```

```
        num_swb = num_swb_long_window[fs_index];
        /* preparation of sect_sfb_offset for long blocks */
        /* also copy the last value! */
        for(i = 0; i < max_sfb + 1; i++) {
            sect_sfb_offset[0][i] = swb_offset_long_window[fs_index][i];
            swb_offset[i] = swb_offset_long_window[fs_index][i];
        }
        break;
    case EIGHT_SHORT_SEQUENCE:
        num_windows = 8;
        num_window_groups = 1;
        window_group_length[num_window_groups-1] = 1;
        num_swb = num_swb_short_window[fs_index];
        for (i = 0; i < num_swb_short_window[fs_index] + 1; i++)
            swb_offset[i] = swb_offset_short_window[fs_index][i];
        for (i = 0; i < num_windows-1; i++) {
            if (bit_set(scale_factor_grouping,6-i)) == 0) {
                num_window_groups += 1;
                window_group_length[num_window_groups-1] = 1;
            }
            else {
                window_group_length[num_window_groups-1] += 1;
            }
        }
        /* preparation of sect_sfb_offset for short blocks */
        for (g = 0; g < num_window_groups; g++) {
            sect_sfb = 0;
            offset = 0;
            for (i = 0; i < max_sfb; i++) {
                width = swb_offset_short_window[fs_index][i+1] -
                          swb_offset_short_window[fs_index][i];
                width *= window_group_length[g];
                sect_sfb_offset[g][sect_sfb++] = offset;
                offset += width;
            }
            sect_sfb_offset[g][sect_sfb] = offset;
        }
        break;
    default:
        break;
}
```

4.5.2.3.5 Order of spectral coefficients in spectral_data

If the Huffman codeword reordering (HCR) tool is used subclause 4.6.16.3.3 applies. Otherwise the spectral coefficients are ordered as follows.

For ONLY_LONG_SEQUENCE windows (num_window_groups = 1, window_group_length[0] = 1) the spectral data is in ascending spectral order, as shown in Figure 4.23.

For the EIGHT_SHORT_SEQUENCE window, the spectral order depends on the grouping in the following manner:

- Groups are ordered sequentially
- Within a group, a scalefactor band consists of the spectral data of all grouped SHORT_WINDOWs for the associated scalefactor window band. To clarify via example, the length of a group is in the range of one to eight SHORT_WINDOWs.
    - If there are eight groups each with length one (num_window_groups = 8, window_group_length[0..7] = 1), the result is a sequence of eight spectra, each in ascending spectral order.
    - If there is only one group with length eight (num_window_groups = 1, window_group_length[0] = 8), the result is that spectral data of all eight SHORT_WINDOWs is interleaved by scalefactor window bands.
    - Figure 4.24 shows the spectral ordering for an EIGHT_SHORT_SEQUENCE with grouping of SHORT_WINDOWs according to Figure 4.22 (num_window_groups = 4).
- Within a scalefactor window band, the coefficients are in ascending spectral order.

4.5.2.3.6 Output word length

The global gain for each audio channel is scaled such that the integer part of the output of the IMDCT can be used directly as a 16-bit PCM audio output to a digital-to-analog (D/A) converter. This is the default mode of operation and will result in correct audio levels. If the decoder has a D/A converter with a resolution greater than 16-bit then the output of the IMDCT can be scaled up such that the appropriate number of fractional bits are included to form the desired D/A word size. In this case the level of the converter output would be matched to that of a 16-bit D/A, but would have the advantage of greater signal dynamic range and lower converter noise floor. Similarly, shorter D/A word lengths can be accommodated.

4.5.2.4 Payloads for the audio object types ER AAC LC, ER AAC LTP, ER AAC LD and ER AAC scalable

For AAC, two kinds of bitstream payload syntax are available: scalable and multichannel. The following changes have to be applied to them to obtain their error resilient counterparts:

- **Multichannel AAC**: The syntax of the top-level payload has been modified. raw_data_block() is replaced by er_raw_data_block() as described in Table 4.19. Please note, that due to this modification coupling_channel_element(), data_stream_element(), program_config_element(), and fill_element() are not supported within the error resilient bitstream payload syntax. For the definition of the helper function bits_to_decode() see 4.5.2.1.
- **Scalable AAC:** The syntax of aac_scalable_main_element() is not changed for error resilience.

No other changes regarding syntax occur.

Data elements are subdivided into different categories depending on its error sensitivity and collected in instances of these categories.

Depending on epConfig, there are several ways to obtain these instances on decoder site as described in subclause 1.6.3.5 of subpart 1.

Both, the order of these instances within an error resilient AAC frame and the order of data elements inside these instances, are described within the next section. If separate access units are used, the dependency structure between elementary streams has to be set up according to the order of instances.

4.5.2.4.1 Error sensitivity category assignment

The following table gives an overview about the error sensitive categories used for AAC (channel_pair_element() = CPE, individual_channel_stream() = ICS, extension_payload() = EPL):

Table 4.80 – Error sensitivity category assignment

category	payload	mandatory	leads / may lead to one instance per	Description
0	main	yes	CPE / stereo layer	commonly used side information
1	main	yes	ICS	channel dependent side information
2	main	no	ICS	error resilient scale factor data
3	main	no	ICS	TNS data
4	main	yes	ICS	spectral data
5	extended	no	EPL	extension type / data_element_version
6	extended	no	EPL	DRC data
7	extended	no	EPL	bit stuffing
8	extended	no	EPL	ANC data
9	extended	no	EPL	SBR data

Table 4.129 shows the category assignment for the main payload (supported elements are SCE, LFE, and CPE). Within this table "-" means that this data element does not occur within this configuration.

Table 4.130 shows the category assignment for the extended payload.

4.5.2.4.2 Category instances and its dependency structure

The subdivision into instances is done on a frame basis, in case of scalable syntax in addition on a layer basis.

The order of instances within the error resilient AAC frame/layer as well as the dependency structure in case of several elementary streams is assigned according to the following rules:

Table 4.81 – Dependency structure within the ER AAC payload

hierarchy level	error resilient multi-channel syntax	error resilience scalable syntax
frame / layer	base payload followed by extension payload	
base payload	order of syntactic elements follows order stated in Table 4.19	commonly used side information followed by channel dependent information
extended payload	no rule regarding the order of multiple EPLs is given, the kind of extension payload can be identified by extension_type	
syntactic element in base payload	commonly used side information followed by channel dependent information	-
channel dependent information	left channel followed by right channel	
channel dependent information / EPL	dependency structure according to category numbers	

Note: Channel dependent information consists of individual_channel_stream() (ICS). Exceptions:

- tp_dlata_present and ltp_data() are treated as channel dependent information even if they are not part of ICS.
- For the ER AAC Scalable object type, tns_data_present, tns_data(), diff_control_data() and diff_control_data_lr() are treated as channel dependent information even if they are not part of ICS.

Figure 4.25 shows an example for the error resilient multi-channel syntax. The order of data elements inside each instance is based on the syntax description, i. e. the logical order is kept. The bits of the data function byte_alignment() terminating the AAC top level payloads are always assigned to the last instance of the dependency chain and stored at its end.

Note: If the reversible variable length coding (RVLC) tool is not used (aacScalefactorDataResilienceFlag == 0), the decoder does not expect instances of error sensitivity category 2, i. e. no access unit shall contain those instances.

4.5.2.5 Payloads for the audio object types TwinVQ and ER TwinVQ

4.5.2.5.1 Definitions

tvq_scalable_main_element()	data element to decode core TwinVQ data.
tvq_scalable_extension_element()	data element to decode scalable TwinVQ data.
vq_single_element()	A data block containing one frame of syntax elements for TwinVQ base layer element and TwinVQ enhancement layer element.
ppc_present	indicates the activation of periodic peak components coding.
postprocess_present	indicates the activation of the postprocessor in the spectral normalization
bandlimit_present	indicates the activation of the bandwidth control of MDCT coefficients
fb_shift	indicates location of active frequency band in enhancement layer
index_blim_h	indicates the upper boundary for the bandwidth control process of the spectrum normalization tool.
index_blim_l	indicates the lower boundary for the bandwidth control process of the spectrum normalization tool
index_shape0	indicates the codevector number of the shape codebook 1 and polarity of the codevector for the interleaved vector quantization tool.
index_shape1	indicates the codevector number of the shape codebook 1 of the interleaved vector quantization tool.
index_env	indicates the codevector number of the Bark-scale envelope codebook of the spectrum normalization tool.
index_fw_alf	indicates the prediction switch of the Bark-scale envelope coding of the spectrum normalization tool.
index_gain	indicates the gain factor of the spectrum normalization tool.
index_gain_sb	indicates the sub-block gain factor of the spectrum normalization tool.

index_lsp0	indicates LSP MA prediction switch of the spectrum normalization tool.
index_lsp1	indicates codevector number of the first-stage LSP VQ in the spectrum normalization tool.
index_lsp2	indicates codevector number of the second-stage LSP VQ in the spectrum normalization tool.
index_shape0_p	indicates the codevector number of codebook 0 of the periodic peak component coding in the spectrum normalization tool.
index_shape1_p	indicates the codevector number of codebook 1 of the periodic peak component coding in the spectrum normalization tool.
index_pit	indicates the base frequency of the periodic peak component in the spectrum normalization tool.
index_pgain	indicates the gain factor of the periodic peak component in the spectrum normalization tool.

4.5.2.5.2 Parameter setting

Following parameters are used for decoding of TwinVQ base layer element and the TwinVQ enhancement layer element:

N_CH	number of channels defined by the system layer
N_DIV	number of sub-vector division for interleaved vector quantization It is calculated according to the decoder status.
N_SF	number of filterbank subblocks in a frame
FW_N_DIV	number of codebook division for the Bark-scale envelop quantization
LSP_SPLIT	number of subvectors for LSP VQ.
N_DIV_P	number of sub-vector division for periodic peak component coding

These parameters are also referenced from the interleaved vector quantization tool and the spectrum normalization tool. The following parameters have constant values as shown below:

Table 4.82 – parameters that have constant values

N_SF (SHORT)	=	8
N_SF (LONG)	=	1
FW_N_DIV	=	7
LSP_SPLIT	=	3
N_DIV_P	=	2*N_CH

4.5.2.5.3 Bit allocation

4.5.2.5.3.1 Spectrum normalization tool

For syntax elements listed below, the number of bits is set according to the parameters **lyr** and **window_sequence**:

Table 4.83 – Bit allocation of syntax elements

Name of variables	Name of number of bits	**lyr** = 0 LONG	**lyr** = 0 SHORT	**lyr** >= 1 LONG	**lyr** >= 1 SHORT	Times per frame
fb_shift	-	0	0	2	2	N_CH
index_blim_h	-	2/0	2/0	0	0	N_CH
index_blim_l	-	1/0	1/0	0	0	N_CH
index_env	FW_N_BIT	6	0	6	0	FW_N_DIV*N_CH

index_fw_alf	-	1	0	1	0	N_CH
index_gain	GAIN_BIT	9	9	8	8	N_CH
index_gain_sb	SUB_GAIN_BIT	0	4	0	4	N_SF*N_CH
index_lsp0	LSP0_BIT	1	1	1	1	N_CH
index_lsp1	LSP1_BIT	6	6	6	6	N_CH
index_lsp2	LSP2_BIT	4	4	4	4	LSP_SPLIT*N_CH
index_shape0_p	MAXBIT_P+1	7/0	0	0	0	N_DIV_P
index_shape1_p	MAXBIT_P+1	7/0	0	0	0	N_DIV_P
index_pit	BASF_BIT	8/0	0	0	0	N_CH
index_pgain	PGAIN_BIT	7/0	0	0	0	N_CH

4.5.2.5.3.2 Interleaved VQ tool

Parameter N_DIV, Number of bits of shape code index 0, bits0, and number of bits of shape code index 1, bits1, are calculated according to the following procedures:

First, number of bits for side information, bits_for_side_information is calculated as follows:

```
bits_for_side_information =
   WS_TBIT + OPT_TBIT + LSP_TBIT + GAIN_TBIT + FW_TBIT + PIT_TBIT+ BAND_TBIT+used_bits
```

where

WS_TBIT Is numer of bits for the flags for window sequence, window shape, joint stereo coding (ms_present), LTP, and TNS tools,

OPT_BIT is number of bits for the flags for bandlimit, ppc, postprocess and active band selection.

Note that If ppc_present Is activated , 43 bits per channel (PIT_TBIT), and bandlimit_present is activated 3 bits per channel (BAND_TBIT) are consumed.

used_bit is number of bits used by tools other than spectrum normalization tool such as joint stereo tools, LTP tools and TNS tools.

LSP_TBIT, GAIN_TBIT, FW_TBIT, and PIT_TBIT is number of bits for optional information, lsp coding, gain coding, Bark-scale envelope coding, and periodic peak components coding respectively. They are set as follows:

```
LSP_TBIT = (LSP_BIT0+LSP_BIT1+(LSP_BIT2*LSP_SPLIT)) * N_CH;
if (FW_N_BIT>0){
   FW_TBIT = ((FW_N_BIT * FW_N_DIV + 1) * N_SF) * N_CH;
}
else{
   FW_TBIT = 0;
}

switch(window_sequence){
case EIGHT_SHORT_SEQUENCE:
   GAIN_TBIT = (GAIN_BIT + SUB_GAIN_BIT * N_SF) * N_CH;
   PIT_TBIT = 0;
   break;
default:
   GAIN_TBIT = GAIN_BIT * N_CH;
   if (ppc_present == TRUE}
      PIT_TBIT = (MAXBIT_P+1)*N_DIV_P*2+ (BASF_BIT + PGAIN_BIT) * N_CH;
   else
      PIT_TBIT = 0;
   break;
}
```

Numbers of bits for side Information are listed in the following table.

Table 4.84 – Bit allocation of side information

Scalable layer **lyr**	0	0	0	0	>=1	>=1	>=1	>=1
Window_sequence	long	long	short	short	long	long	short	short

N_CH	1	2	1	2	1	2	1	2
WS_TBIT	5	9	5	9	0	2	0	2
OPT_TBIT	3	3	1	1	2	4	2	4
LSP_TBIT	19	38	19	38	19	38	19	38
GAIN_TBIT	9	18	41	82	8	16	40	80
FW_TBIT	43	86	0	0	43	86	0	0
BAND_TBIT	3/0	6/0	3/0	6/0	0	0	0	0
PIT_TBIT	43/0	86/0	0	0	0	0	0	0

Number of available bits, bits_available_vq is calculated as follows:

bits_available_vq = (int)(((FRAME_SIZE * bitrate/sampling_frequency)/8+0.5)*8) - bits_for_side_information,

where bitrate is given by a system parameter in [bit/s] and sampling frequency is given in the right column of Table 4.68.

Finally, number of shape sub-vector, N_DIV and number of bits for shape code indices, bits0 and bits1, are calculated as follows:

```
N_DIV = ((int)((bits_available_vq + MAXBIT*2-1)/(MAXBIT*2)));
bits = (bits_available_vq + N_DIV - 1 - idiv) / N_DIV;
bits0 = (int)(bits+1) / 2;
bits1 = (int)bits/2;
```

where MAXBIT is maximum number of shape code bit. The MAXBIT is always set to 6.

4.5.2.5.3.3 Scalable coder by means of TwinVQ

Scalable coder can be constructed by cascading the TwinVQ core and the TwinVQ extension (enhancement) quantizers. The decoding process is straight forward. MDCT coefficients are reconstructed by using the demultiplexed bitstream payload for each layer. Each enhancement layer covers the different part of the coefficients whose regions are adaptively specified by the shift parameters. Summation of these reconstructed coefficients is performed. Finally, an IMDCT filter bank maps back the spectrum (MDCT coefficients) into a time domain signal using a synthesis window and overlap / add techniques. If one wishes to decode only the core MDCT coefficients, the decoded core coefficients have to be passed on directly to the filter bank. The core layer of this scalable coder can be combined with the AAC tools such as TNS, LTP and joint stereo coding. This coder, however, does neither support the mono-stereo cascading coder nor the FSS control.

If in tvq_scalable_element() the following condition is true:

(**ms_mask** == 1) && (**this_layer_stereo** == 1) &&
(**window_sequence** == EIGHT_SHORT_SEQUENCE) &&

there is no **scale_factor_grouping** information in the previous layer(s)

then

num_window_group shall be set to ‘1’ (scale_factor_grouping = 0x7F).

4.5.2.6 Payloads for the audio object type ER BSAC

4.5.2.6.1 Decoding of payload for audio object type ER BSAC (bsac_ payload())

Fine grain scalability would create large overhead if one would try to transmit fine grain layers over multiple elementary streams (ES). So, in order to reduce overhead and implement the fine grain scalability efficiently in current MPEG-4 system, the server can organize the fine grain audio data into the payload by dividing the fine grain audio data into the large-step layers and concatenating the large step layers of the several sub-frames. Then the payload is transmitted over ES.

So, the payload transmitted over ES requires the rearrangement process for the actual decoding.

4.5.2.6.1.1 Definitions

bsac_payload(lay)	Sequnece of bsac_lstep_element()s. Syntactic element of the payload transmitted over layth layer ES. A bsac_payload(lay) basically consists of several layth layer bitstream, bsac_lstep_element() of serveral sub-frames.
bsac_lstep_element(frm, lay)	Syntactic element for the layth large-step layer bitstream of frmth sub-frame.
bsac_stream_byte[frm][offset+i]	(offset+i)-th byte which is extracted from the payload. After bsac stream bytes are extracted from all the payloads that have been transmitted to the receiver, these data are concatenated and saved in the array bsac_stream_byte[frm][] which is the bitstream of frmth sub-frame. Then, we proceed to decode the concatenated stream,bsac_stream_byte[frm][] using the syntax of the BSAC fine grain scalability.

4.5.2.6.1.1.1 Help elements

data_available()	function that returns “1” as long as data is available, otherwise “0”
LayerStartByte[frm][lay]	Start position of layth large-step layer in bytes which is located on frmth sub-frame’s bitstream. See sub-clause 4.5.2.6.1.2 for the calculation process of this value.
LayerLength[frm][lay]	Length of the large-step layer in bytes which is located on the payload of layth layer ES and concatenated to frmth sub-frame’s bitstream. See sub-clause 4.5.2.6.1.2 for the calculation process of this value.
LayerOffset[frm][lay]	Start position of the large-step layer of frmth frame in bytes which is located on the payload of layth layer ES. See sub-clause 4.5.2.6.1.2 for the calculation process of this value.
frm	index of frame in which bsac stream bytes are saved.
lay	index of the large-step layer over which the fine granule audio data is transmitted.
numOfSubFrame	Number of the sub-frames which are grouped and transmitted in a super-frame in order to reduce the transmission overhead.
layer_length	Average length of the large-step layers in bytes which are assembled in a payload.
numOfLayer	number of the large-step layers which the fine grain audio data is divided into.

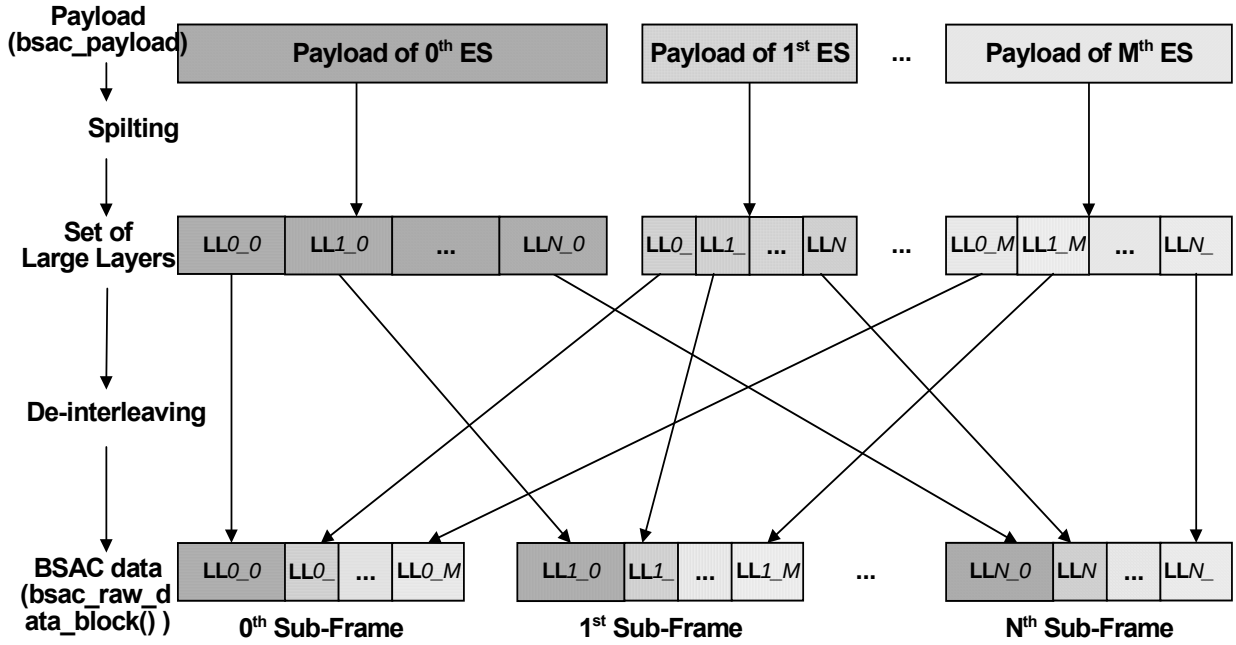
4.5.2.6.1.2 Decoding process

On the sync layer (SL) of MPEG-4 system, an elementary stream is packetized into access units or parts thereof. Such a packet is called SL packet. Access Unit(AU)s are the only semantic entities at the sync layer (SL) of MPEG-4 system that need to be preserved from end to end. AUs are used as the basic unit for synchronization which are made up of one or more SL packets.

The dynamic data for the BSAC is transmitted as SL_Packet payload in the base layer Elementary Stream(ES) and the enhancement layer ESs. The dynamic data is made up of the large-step layers of one or more subsequent sub-frames.

When the SL packets of an AU arrives in the receiver, a sequence of packet is mapped into a payload which is split into the large step layers, bsac_lstep_layer(frm, lay) for the subsequent sub-frames. And the split layers should be concatenated with the large-step layers which are transmitted over the other ES.

In the receiver, BSAC data is reconstructed from the payloads as shown in Figure 4.16.



where, **LL*i*_*k*** is the k-th large-step layer of the i-th sub-frame

(M+1) is the number of the large-step layer to be transmitted (numOfLayer)

(N+1) is the number of the sub-frame to be grouped in an AU (numOfSubFrame)

Figure 4.16 – Reconstruction of BSAC data

The large-step layers are split from a payload of k^{th} layer ES that is organized as shown in Figure 4.17.

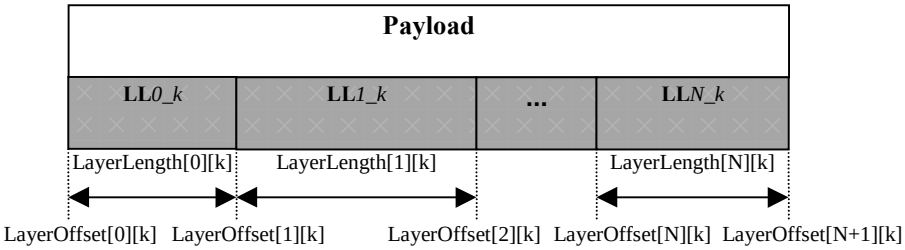


Figure 4.17 – Structure of the payload

The split large-step layers are deinterleaved and concatenated to map the entire fine grain BSAC data. And then, decode the concatenated bitstreams using the syntax (bsac_raw_data_block()) for fine grain scalability to make the reconstructed signal.

Some help variables and arrays are needed to describe the re-arranging process of the payload transmitted over ES. These help variables depend on layer, numOfLayer, **numOfSubFrame**, **layer_length** and **frame_length** and must be built up for mapping bsac_raw_data_block() of each sub-frame from the payloads. The pseudo code shown below describes

- how to calculate *LayerLength[i][k]*, the length of the large-step layer which is located on the fine granule audio data, bsac_raw_data_block() of i^{th} sub-frame.
- how to caluculate *LayerOffset[i][k]* which indicates the start position of the large-step layer of i^{th} frame which is located on the payload of the k^{th} ES (bsac_payload())
- how to calculate *LayerStartByte[i][k]* which indicates the start position of the large-step layer which is located on the fine granule audio data, bsac_raw_data_block() of i^{th} sub-frame

```
for (k = 0; k < numOfLayer; k ++) {
   LayerStartByte[0][k] = 0;
   for (i = 0; i < numOfSubFrame; i++) {
      if (k == (numOfLayer-1) ) {
         LayerEndByte[i][k] = frame_length[i];
      } else {
         LayerEndByte[i][k] = LayerStartByte[i][k] + layer_length[k];
         if (frame_length[i] < LayerEndByte[i][k])
            LayerEndByte [i][k] = frame_length[i];
      }
      LayerStartByte[i+1][k] = LayerEndByte[i][k];
      LayerLength[i][k] = LayerEndByte[i][k] - LayerStartByte[i][k];
   }
}
for (k = 0; k < numOfLayer; k ++) {
   LayerOffset[0][k] = 0;
   for (i = 0; i < numOfSubFrame; i++) {
      LayerOffset[i+1][k] = LayerOffset[i][k] + LayerLength[i][k];
   }
}
```

Where, *frame_length[i]* is the length of i^{th} frame's bitstream which is obtained from the syntax element **frame_length** and layer_length[i] is the average length of the large-step layers in the payload of i^{th} layer ES and is obtained from Audio DecoderSpecificInfo.

4.5.2.6.2 Decoding of a bsac_raw_data_block()

4.5.2.6.2.1 Definitions

4.5.2.6.2.1.1 Data elements

bsac_raw_data_block()	block of raw data that contains coded audio data, related information and other data. A bsac_raw_data_block() basically consists of bsac_base_element() and several bsac_layer_element().
bsac_base_element()	Syntactic element of the base layer bitstream containing coded audio data, related information and other data.
frame_length	the length of the frame including headers in bytes.
bsac_header()	contains general information used for BSAC.
header_length	the length of the headers including frame_length, bsac_header() and general_header() in bytes. The actual length is (header_length+7) bytes. However if header_length is 0, it represents that the actual length is smaller than or equal to 7 bytes. And if header_length is 15, it represents that the actual length is larger than or equal to (15+7) bytes and should be calculated through the decoding of the headers .
sba_mode	indicates that the segmented binary arithmetic coding (SBA) scheme is used if this element is 1. Otherwise the general binary arithmetic coding scheme is used.
top_layer	top scalability layer index
base_snf_thr	significance threshold used for coding the bit-sliced data of the base layer.
base_band	indicates the maximum spectral line of the base layer. If the window_sequence is SHORT_WINDOW, 4*base_band is the maximum spectral line. Otherwise, 32*base_band is the maximum spectral line.
max_scalefactor[ch]	the maximum value of the scalefactors
cband_si_type[ch]	the type of the coding band side-information(si). Using this element, the largest value of cband_si's and the arithmetic model for decoding cband_si can be set as shown in Table 4.A.29.

base_scf_model[ch]	the arithmetic model for decoding the scalefactors in the base layer.
enh_scf_model[ch]	the arithmetic model used for decoding the scalefactors in the other enhancement layers.
max_sfb_si_len[ch]	maximum length which can be used per channel for coding the scalefactor-band side information including scalefactor and stereo-related information within a scalefactor band. This value has a offset(5). The actual maximum length is (max_sfb_si_len+5). This value is used for determining the bitstream size of each layer.
general_header()	contains header data for the General Audio Coding
reserved_bit	bit reserved for future use
window_sequence	indicates the sequence of windows. See ISO/IEC 14496-3 General Audio Coding.
window_shape	A 1 bit field that determines what window is used for the trailing part of this analysis window
max_sfb	number of scalefactor bands transmitted per group
scale_factor_grouping	A bit field that contains information about grouping of short spectral data
pns_data_present	the flag indicating whether the perceptual noise substitution(pns) will be used (1) or not (0).
pns_start_sfb	the scalefactor band from which the pcns tool is started.
ms_mask_present	this two bit field (see) indicates that the stereo mask is 00 Independent 01 1 bit mask of ms_used is located in the layer sfb side information part (layer_sfb_si()). 10 All ms_used are ones 11 2 bit mask of stereo_info is located in the layer sfb side information part layer_sfb_si()).
layer_cband_si()	contains the coding band side information necessary for Arithmetic encoding/decoding of the bit-sliced data within a coding band.
layer_sfb_si()	contains the side information of a scalefactor band such as the stereo-, the pns and the scalefactor information.bsac_layer_element() Syntactic element of the enhancement layer bitstream containing coded audio data for a time period of 1024(960) samples, related information and other data.
bsac_layer_spectra()	contains the arithmetic coded audio data of the quantized spectral coefficients which are newly added to each layer. See subclause 4.5.2.6.2.5 for the new spectral coefficients.
bsac_lower_spectra()	contains the arithmetic coded audio data of the quantized spectral coeffients which are lower than the spectra added to each layer.
bsac_higher_spectra()	contains the arithmetic coded audio data of the quantized spectral coeffients which are higher than the spectra added to each layer.
bsac_spectral_data()	contains the arithmetic coded audio data of the quantized spectral coeffients.

4.5.2.6.2.1.2 Help elements

data_available()	function that returns “1” as long as bitstream is available, otherwise “0”
nch	a data element that identifies the number of the channel.
scalefactor window band	term for scalefactor bands within a window. See ISO/IEC 14496-3 General Audio Coding.
scalefactor band	term for scalefactor band within a group. In case of EIGHT_SHORT_SEQUENCE and grouping a scalefactor band may

	contain several scalefactor window bands of corresponding frequency. For all other window_sequences scalefactor bands and scalefactor window bands are identical.
g	group index
win	window index within group
sfb	scalefactor band index within group
swb	scalefactor window band index within window
num_window_groups	number of groups of windows which share one set of scalefactors. See subclause 4.5.2.6.2.4
window_group_length[g]	number of windows in each group. See subclause 4.5.2.6.2.4
bit_set(bit_field,bit_num)	function that returns the value of bit number bit_num of a bit_field (most right bit is bit 0)
num_windows	number of windows of the actual window sequence. See subclause 4.5.2.6.2.4
num_swb_long_window	number of scalefactor bands for long windows. This number has to be selected depending on the sampling frequency. See ISO/IEC 14496-3 General Audio Coding.
num_swb_short_window	number of scalefactor window bands for short windows. This number has to be selected depending on the sampling frequency. See ISO/IEC 14496-3 General Audio Coding.
num_swb	number of scalefactor window bands for shortwindows in case of EIGHT_SHORT_SEQUENCE, number of scalefactor window bands for long windows otherwise. See subclause 4.5.2.6.2.4
swb_offset_long_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for long windows. This table has to be selected depending on the sampling frequency. See ISO/IEC 14496-3 General Audio Coding.
swb_offset_short_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows. This table has to be selected depending on the sampling frequency. See ISO/IEC 14496-3 General Audio Coding.
swb_offset[g][swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows. See subclause 4.5.2.6.2.4
layer_group[layer]	indicates the group index of the spectral data to be added newly in the scalability layer
layer_start_sfb[layer]	indicates the index of the lowest scalefactor band index to be added newly in the scalability layer
layer_end_sfb[layer]	indicates the highest scalefactor band index to be added newly in the scalability layer
layer_start_cband[layer]	indicates the lowest coding band index to be added newly in the scalability layer
layer_end_cband[layer]	indicates the highest coding band index to be added newly in the scalability layer
layer_start_index[layer]	indicates the index of the lowest spectral component to be added newly in the scalability layer
layer_end_index[layer]	indicates the index of the highest spectral component to be added newly in the scalability layer
start_index[g]	indicates the index of the lowest spectral component to be coded in the group g

end_index[g]	indicates the index of the highest spectral component to be coded in the group g
layer_data_available()	function that returns “1” as long as each layer’s bitstream is available, otherwise “0”. In other words, this function indicates whether the remaing bitstream of each layer is available or not.
terminal_layer[layer]	indicates whether a layer is the terminal layer of a segment which is made up of one or more scalability layers. If the segmented binary arithmetic coding is not activated, all these values are always set to 0 except that of the top layer. Otherwise, these values are defined as described in subclause 4.6.4.6.3.

4.5.2.6.2.2 Decoding process

4.5.2.6.2.2.1 Zero stuffing

In order to do the arithmetic decoding perfectly, 32-bit zero value should be concatenated to the bitstream. In case of the SBA mode, the bitstream of a frame is split into the several segments. So, zero value should be concatenated to all segments. For the detailed description, see subclause 4.6.4.6.3. However, in case of the non-SBA mode, one zero stuffing is good enough since the bitstream of a frame is not split.

4.5.2.6.2.2.2 bsac_raw_data_block

A total BSAC stream, bsac_raw_data_block has the layered structure. First, bsac_base_element() is parsed and decoded which is the bitstream for base scalability layer. Then, bsac_layer_element() for the next enhancement layer is parsed and decoded. bsac_layer_element() decoding routine is repeated while the decoded bitstream data is available and layer is smaller than or equal to the top layer, **top_layer.**

4.5.2.6.2.2.3 bsac_base_element

A bsac_base_element() is made up of **frame_length**, bsac_header, general_header and bsac_layer_element().

First, **frame_length** is parsed from syntax. It represents the length of the frame including headers in bytes.

The syntax elements for the base layer are parsed which are composed of a bsac_header(), a general_header(), a layer_cband_si(), layer_sfb_si() and bsac_layer_element. bsac_base_element() has several bsac_layer_element() because the base layer is split into the several sub-layers for the error resilience of the base layer. The number of the sub-layers, *slayer_size* is calculated using the group index and the coding band as shown in subclause 4.5.2.6.2.5.

4.5.2.6.2.2.4 Recovering a bsac_header

BSAC provides a 1-kits/sec/ch fine grain scalability which has the layered structure, one base layer and several enhancement layers. Base layer contains the general side information for all the layers, the specific side information for the base layer and the audio data. The general side information is transmitted in the syntax of bsac_header() and general_header().

bsac_header consists of **top_layer**, **header_length**, **sba_mode, base_band**, **max_scalefactor**, **cband_si_type**, **base_scf_model** and **enh_scf_model**. All the data elements are included in the form of the unsigned integer.

First, 4 bit **header_length** is parsed which represents the length of the headers including frame_length, bsac_header and general_header in bytes. The length of the headers is (header_length+7)*8. Next, 1bit **sba_mode** is parsed which resprents whether the segmented binary arithmetic coding(SBA) is used or the binary arithmetic coding is used.

Next, 6 bit **top_layer** is parsed which represents the top scalability layer index to be encoded. Next, 2 bit **base_snf_thr** is parsed which represents the significance threshold used for coding the bit-sliced data of the base layer.

Next, 8 bit **max_scalefactor** is parsed which represents the maximum value of the scalefactors. If the number of the channel is not 1, this value is parsed one more.

Next, 5-bit **base_band** is parsed which represents minimum spectral line which is coded in the base layer. If the window sequence is SHORT_WINDOW, 4*base_band indicates the minimum spectral line. Otherwise 32*base_band indicates the minimum spectral line.

And, 5 bit **cband_si_type** is parsed which represents the arithmetic model of cband_si and the largest cband_si which can be decoded as shown in Table 4.A.29. 3 bit **base_scf_model** and **enh_scf_model** are parsed which represent the arithmetic model table for the scalefactors of the base layer and the other enhancement layers, respectively. Next, 4 bit **max_sfb_si_len** is pared which represents the maximum length of the scalefactor band side information to be able to used in each layer. The maximum length is (max_sfb_si_len+5).

4.5.2.6.2.2.5 Recovering a general_header

The order for decoding the syntax of a bsac_header is:

- get reserved_bit
- get window_sequence
- get window_shape
- get max_sfb
- get scale_factor_grouping if the window_sequence is EIGHT_SHORT_SEQUENCE
- get pns_present
- get pns_start_sfb if present
- get ms_mask_present flag if the number of the channel is 2
- get tns_data_present
- get TNS data if present
- get ltp_data_present
- get ltp data if present

If the number of the channel is not 1, the decoding of another channel is done as follows:

- get tns_data_present
- get TNS data if present
- get ltp_data_present
- get ltp data if present

The process of recovering tns_data and ltp_data is described in ISO/IEC 14496-3 General Audio Coding.

4.5.2.6.2.2.6 bsac_layer_element

A bsac_layer_element() is an enhancement layer bitstream and composed of layer_cband_si(), layer_sfb_si(), bsac_layer_spectra(), bsac_lower_spectra() and bsac_higher_spectra(). Decoding process of bsac_layer_element() is as follows:

- Decode layer_cband_si
- Decode layer_sfb_si
- Decode bsac_layer_spectra
- Decode bsac_lower_spectra
- Decode bsac_higher_spectra

4.5.2.6.2.2.7 Decoding of coding band side information (layer_cband_si)

The spectral coefficients are divided into coding bands which contain 32 quantized spectral coefficients for the noiseless coding. Coding bands(abbreviation ‘cband’) are the basic units used for the noiseless coding.

cband_si represents the MSB plane and the probability table of the sliced bits within a coding band as shown in Table 4.A.31. Using this cband_si, the bit-sliced data of each coding band are arithmetic-coded.

cband_si is arithmetic_coded with the model which is given in the syntax element **cband_si_type** as shown in Table 4.A.29.

An overview of how to decode cband_si will be given in subclause 4.6.4.5.

4.5.2.6.2.2.8 Decoding of the scalefactor band side information (layer_sfb_si)

An overview of how to decode layer_sfb_si will be given here. layer_sfb_si is made up of as follows:

Decoding of stereo_info, ms_used or noise_flag.

Decoding of scalefactors

4.5.2.6.2.2.9 Decoding of stereo_info, noise_flag or ms_used

Decoding process of stereo_info, noise_flag or ms_used is depended on pns_data_present, number of channel, ms_mask_present.

If pns data is not present, decoding process is as follows:

If ms_mask_present is 0, arithmetic decoding of stereo_info or ms_used is not needed.

If ms_mask_present is 2, all ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band.

If ms_mask_present is 1, 1 bit mask of max_sfb bands of ms_used is conveyed in this case. So, ms_used is arithmetic decoded. M/S stereo processing of AAC is done according to the decoded ms_used.

If ms_mask_present is 3, stereo_info is arithmetic decoded. stereo_info is two-bit flag per scalefactor band indicating the M/S coding or Intensity coding mode. If stereo_info is not 0, M/S stereo or intensity stereo of AAC is done with these decoded data.

If pns data is present and the number of channel is 1, decoding process is as follows:

If the number of channel is 1 and pns data is present, noise flag of the scalefactor bands between **pns_start_sfb** to **max_sfb** is arithmetic decoded. Perceptual noise substitution is done according to the decoded noise flag.

If pns data is present and the number of channel is 2, decoding process is as follows:

If ms_mask_present is 0, noise flag for pns is arithmetic decoded. Perceptual noise substitution of independent mode is done according to the decoded noise flag.

If ms_mask_present is 2, all ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band. However, there is no pns processing regardless of pns_data_present flag

If ms_mask_present is 1, 1 bit mask of max_sfb bands of ms_used is conveyed in this case. So, ms_used is arithmetic decoded. M/S stereo processing of AAC is done according to the decoded ms_used. If ms_used is 1, there is no pns processing.

If ms_mask_present is 3, stereo_info is arithmetic decoded. If stereo_info is 1 or 2, M/S stereo or intensity stereo processing of AAC is done with these decoded data and there is no pns processing. If stereo_info is 3 and scalefactor band is smaller than pns_start_sfb, out_of_phase intensity stereo processing is done. If stereo_info is 3 and scalefactor band is larger than or equal to pns_start_sfb, noise flag for pns is arithmetic decoded. And then if the both noise flags of two channel are 1, noise substitution mode is arithmetic decoded. The perceptual noise is substituted or out_of_phase intensity stereo processing is done according to the substitution mode. Otherwise, the perceptual noise is substituted only if noise flag is 1.

The detailed description of how to decode this side information will be given in subclause 4.6.4.3.

4.5.2.6.2.2.10 Decoding of scalefactors

The spectral coefficients are divided into scalefactor bands that contain a multiple of 4 quantized spectral coefficients. Each scalefactor band has a scalefactor. For all scalefactors the difference to the maximum scalefactor value, **max_scalefactor** is arithmetic-coded using the arithmetic model given in Table 4.A.30. The arithmetic model necessary for coding the differential scalefactors in the base layer is given as a 3-bit unsigned integer in the data element, **base_scf_model**. The arithmetic model necessary for coding the differential scalefactors in the other enhancement layers is given as a 3-bit unsigned integer in the data element, **enh_scf_model**. The maximum scalefactor value is given explicitly as a 8 bit PCM in the data element **max_scalefactor**.

The detailed description of how to decode this side information will be given in subclause 4.6.4.4.

4.5.2.6.2.2.11 Bit-Sliced spectral data

In BSAC encoder, the absolute values of quantized spectral coefficients is mapped into a bit-sliced sequence. These slicded bits are the symbols of the arithmetic coding. Every siliced bits are binary arithmetic coded with the proper probability (arithmetic model) from the lowest-frequency coefficient to the highest-frequency coefficient of the scalability layer, starting the Most Significant Bit(MSB) plane and progressing to the Least Significant Bit(LSB) plane. The arithmetic coding of the sign bits associated with non-zero coefficient follows that of the sliced bit when the sliced bit is 1 for the first time.

The probability value should be defined in order to arithmetic-code the symbols (the sliced bits). Binary probability table is made up of probability values of the symbol “0”. First of all, probability table is selected using cband_si as shown Table 4.A.31. The probability value is selected among the several values in the selected table according to the context such as the remaining available bit size and the sliced bits of successive non-overlapping 4 spectral data.

For the case of multiple windows per block, the concatenated and possibly grouped and interleaved set of spectral coefficients is treated as a single set of coefficients that progress from low to high as described in subclause 4.5.2.6.2.6. This set of spectral coefficients needs to be de-interleaved after they are decoded. The set of bit-sliced sequence is divided into coding bands. The probability table index used for encoding the bit-sliced data within each coding band is included in the bitsteam element **cband_si** and transmitted starting from the lowest frequency coding band and progressing to the highest frequency coding band. The spectral information for all scalefactor bands equal to or greater than **max_sfb** is set to zero.

4.5.2.6.2.2.12 Decoding the sliced bits of the spectral data

The spectral bandwidth is increased in proportion to the scalability layer. So, the new spectral data is added to each layer. First of all, these new spectral data are coded in each layer (bsac_layer_spectra()). The coding process is continued until the data of each layer is not available or all the sliced bits of the new spectra are coded. The length of the available bitstream (*available_len[]*) is initialized at the beginning of each layer as described in subclause 4.5.2.6.2.5. The estimated length of the codeword (*est_cw_len*) to be decoded is calculated from the arithmetic decoding process as described in subclause 4.5.2.6.2.7. After the arithmetic decoding of a symbol, the length of the available bitstream should be updated by subtracting the estimated codeword length from it. We can detect whether the remaing bitstream of each layer is available or not by checking the array *available_len[]*.

From the lowest layer to the top layer, the new spectra are arithmetic-coded layer-by-layer in the above first process (bsac_layer_spectra()). Some sliced bits cannot be coded for lack of the codeword allocated to the layer. After the first coding process is finished, the current significances (cur_snf) are saved for the secondary coding processes (bsac_lower_spectra() and bsac_higher_spectra()). The sliced bits which remain uncoded is coded using the saved significances(unc_snf) in the secondary coding process.

If there remains the available codewords after the first coding, the next symbol to be decoded with these redundant codewords depends on whether the segmented binary arithmetic coding(SBA) mode is active.

In case of the non-SBA mode, the uncoded symbols of the lower spectra in the layers than the current layer are coded in the secondary coding process (bsac_lower_spectra()).

In case of the SBA mode for the error resilience, the next symbol is dependant upon whether the layer is the terminal layer of a segment or not. If the layer is not a terminal of the segment, the spectral data of the next layer (*bsac_layer_spectra(layer+1)*) should be decoded. That is to say, the redundant length of the layer is added to the available bitstream length (*available_len[layer+1]*) of the next layer in the first coding process.

If the layer is a terminal of the segment, the uncoded symbols of the lower spectra in the layers than the current layer are coded in the secondary coding process (bsac_lower_spectra()). The uncoded symbol of the spectra in the layers higher than the current layer are coded in the secondary coding process (bsac_higher_spectra()) if the codeword of the layer is available in spite of having coded the lower spectra. And the remaining symbols are continuously coded in the layers whose codeword is available, starting from the lowest layer and progressing to the top layer.

If there are the redundant bits after the secondary coding, the size of the redundant bits is added to the available bitstream length(*available_len[layer+1]*) of the next layer and the redundant bits are used in the first coding of the next layer.

4.5.2.6.2.2.13 Reconstruction of the decoded sample from bit-sliced data

In order to reconstruct the spectral data, a bit-sliced sequence that has been decoded should be mapped into quantized spectral values. An arithmetic decoded symbol is a sliced bit. A decoded symbol is translated to the bit values of quantized spectral coeffiecients, as specified in the following pseudo C code:

```
snf = the significance of the symbol (the sliced bit) to be decoded;
sliced_bit[ch][g][i][snf] = the decoded symbol (the sliced bits of the quantized spectrum);
sample[ch][g][i] = buffer for quantized spectral coeffients to be recontructed;
scaled_bit = sliced_bit[ch][g][i][snf] << (snf-1);
if (sample[ch][g][i] < 0)
    sample[ch][g][i] -= scaled_bit;
else
    sample[ch][g][i] += scaled_bit;
```

And if the sign bit of the decoded sample is 1, the decoded sample sample[ch][g][i] has the negative value as follows :

```
if (sample[ch][g][i] != 0) {
    if (sign_bit[ch][g][i] == 1)     sample[ch][g][i] = -sample[ch][g][i];
    }
```

4.5.2.6.2.3 Windows and window sequences for BSAC

Quantization and coding is done in the frequency domain. For this purpose, the time signal is mapped into the frequency domain in the encoder. Depending on the signal, the coder may change the time/frequency resolution by using two different windows: LONG_WINDOW and SHORT_WINDOW. To switch between windows, the transition windows LONG_START_WINDOW and LONG_STOP_WINDOW are used. Refer to ISO/IEC 14496-3 General Audio Coding for more detailed information about the transform and the windows since BSAC has the same transform and windows with AAC.

4.5.2.6.2.4 Scalefactor bands, grouping and coding bands for BSAC

Many tools of the AAC/BSAC decoder perform operations on groups of consecutive spectral values called scalefactor bands (abbreviation “sfb”). The width of the scalefactor bands is built in imitation of the critical bands of the human auditory system. For that reason the number of scalefactor bands in a spectrum and their width depend on the transform length and the sampling frequency. Refer to ISO/IEC 14496-3 General Audio Coding for more detailed information about the scalefactor bands and grouping because BSAC has the same process with AAC.

BSAC decoding tool performs operations on groups of consecutive spectral values called coding bands (abbreviation “cband”). To increase the efficiency of the noiseless coding, the width of the coding bands is fixed as 32 irrespective of the transform length and the sampling frequency. In case of sequences which contain LONG_WINDOW, 32 spectral data are simply grouped into a coding band. Since the spectral data within a group are interleaved in an ascending spectral order in case of SHORT_WINDOW, the interleaved spectral data are grouped into a coding band. Each spectral index within a group is mapped into a coding band with a mapping function, *cband* = spectral_index/32.

Since scalefactor bands and coding bands are a basic element of the BSAC coding algorithm, some help variables and arrays are needed to describe the decoding process in all tools using scalefactor bands and coding bands. These help variables must be defined for BSAC decoding. These help variables depend on sampling_frequency, **window_sequence, scalefactor_grouping** and **max_sfb** and must be built up for each bsac_raw_data_block. The pseudo code shown below describes

- how to determine the number of windows in a window_sequence *num_windows*
- how to determine the number of window_groups *num_window_groups*
- how to determine the number of windows in each group *window_group_length[g]*
- how to determine the total number of scalefactor window bands *num_swb* for the actual window type
- how to determine *swb_offset[g][swb]*, the offset of the first coefficient in scalefactor window band *swb* of the window actually used

A long transform window is always described as a window_group containing a single window. Since the number of scalefactor bands and their width depend on the sampling frequency, the affected variables are indexed with sampling_frequency_index to select the appropriate table.

```
fs_index = sampling_frequency_index;
```

```
switch (window_sequence) {
   case ONLY_LONG_SEQUENCE:
   case LONG_START_SEQUENCE:
   case LONG_STOP_SEQUENCE:
      num_windows = 1;
      num_window_groups = 1;
      window_group_length[num_window_groups-1] = 1;
      num_swb = num_swb_long_window[fs_index];
      for (sfb = 0; sfb < max_sfb+1; sfb++) {
            swb_offset[0][sfb] = swb_offset_long_window[fs_index][sfb];
      }
      break;
   case EIGHT_SHORT_SEQUENCE:
      num_windows = 8;
      num_window_groups = 1;
      window_group_length[num_window_groups-1] = 1;
      num_swb = num_swb_short_window[fs_index];
      for(i = 0; i < num_windows-1; i++) {
         if (bit_set(scale_factor_grouping,6-i)) == 0 ) {
            num_window_groups += 1;
            window_group_length[num_window_groups-1] = 1;
         }
         else {
            window_group_length[num_window_groups-1] += 1;
         }
      }

      for (g = 0; g < num_window_groups; g++)
         swb_offset[g][0] = 0;

      for (sfb = 0; sfb < max_sfb+1; sfb++) {
         for (g = 0; g < num_window_groups; g++) {
            swb_offset[g][sfb] = swb_offset_short_window[fs_index][sfb];
            swb_offset[g][sfb] = swb_offset[g][sfb] * window_group_length[g];
         }
      }
      break;
   default:
      break;
}
```

4.5.2.6.2.5 BSAC fine grain scalability layer

BSAC provides a 1-kits/sec/ch fine grain scalability which has the layered bitstream, one BSAC base layer and various enhancement layers. BSAC base layer is made up of the general side information for all the fine grain layers, the specific side information for only the base layer and the audio data. BSAC enhancement layers contain the layer side information and the audio data.

BSAC scalable coding scheme has the scalable band-limit according to the fine grain layer. First of all, the base band-limit is set. The base band-limit depends on the signal to be encoded and is in the syntax element, **base_band**. The actually limited spectral line is 4*base_band if the window sequence is SHORT_WINDOW. Otherwise, the limited spectral line is 32*base_band. In order to provide the fine grain scalability, BSAC extends the band-limit according to the fine grain layer. The band limit of each layer depends on the base band-limit, the transform lengths 1024(960) and 128(120) and the sampling frequencies. The spectral band is extended more and more as the number of the enhancement layer is increased. So, the new spectral components are added to each layer.

Some help variables and arrays are needed to describe the bit-sliced decoding process of the side information and spectral data in each BSAC fine grain layer. These help variables depend on sampling_frequency, layer, **nch, frame_length, top_layer, window_sequence** and **max_sfb** and must be built up for each bsac_layer_element. The pseudo code shown below describes

- how to determine *slayer_size*, the number of the sub-layers which the base layer is split into.

```
slayer_size = 0;
```

```
for (g = 0; g < num_window_groups; g++) {
    if (window_sequence == EIGHT_SHORT_SEQUENCE) {
        end_index[g] = base_band * 4 * window_group_length[g];
        if (fs == 44100 || fs == 48000) {
            if (end_index[g]%32>=16)
                end_index[g] = (int)(end_index[g]/32)*32 + 20;
            else if (end_index[g]%32 >= 4)
                end_index[g] = (int)(end_index[g]/32)*32 + 8;
        }
        else if (fs == 22050 || fs == 24000 || fs == 32000)
            end_index[g] = (int)(end_index[g]/16)*16;
        else if (fs == 11025 || fs == 12000 || fs == 16000)
            end_index[g] = (int)(end_index[g]/32)*32;
        else end_index[g] = (int)(end_index[g]/64)*64;
        end_cband[g] = (end_index[g] + 31) / 32;
    }
    else
        end_cband[g] = base_band;
    slayer_size += end_cband[g];
}
```

- how to determine *layer_group[]*, the group index of the spectral components to be added newly in the scalability layer

```
layer = 0
for (g = 0; g < num_window_groups; g++)
for (cband = 1; cband <= end_cband[g]; layer++, cband++)
    layer_group[layer] = g;

layer = slayer_size;
```

- how to determine *layer_end_index[]*, the end offset of the spectral components to be added newly in each scalability layer
- how to determine *layer_end_cband[]*, the end coding band to be added newly in each scalability layer
- how to determine *layer_start_index[]*, the start offset of the spectral components to be added newly in each scalability layer
- how to determine *layer_start_cband[]*, the start coding band to be added newly in each scalability layer

```
layer = 0;
for (g = 0; g < num_window_groups; g++) {
        for (cband = 0; cband < end_cband[g]; cband++) {
            layer_start_cband[layer] = cband;
            end_cband[g] = layer_end_cband[layer] = cband+1;
            layer_start_index[layer] = cband * 32;
            end_index[g] = layer_end_index[layer++] = (cband+1) * 32;
        }
        if (window_sequence == EIGHT_SHORT_SEQUENCE)
            last_index[g] = swb_offset_short_window[fs_index][max_sfb] * window_group_length[g];
        else
            last_index[g] = swb_offset_long_window[fs_index][max_sfb];
}

for (layer = slayer_size; layer < (top_layer+slayer_size); layer++) {
    g = layer_group[layer];
    layer_start_index[layer] = end_index[g];
    if (fs == 44100 || fs == 48000) {
        if (end_index[g]%32 == 0)
            end_index[g] += 8;
        else
            end_index[g] += 12;
    }
    else if (fs == 22050 || fs == 24000 || fs == 32000)
```

```
        end_index[g] += 16;
    else if (fs == 11025 || fs == 12000 || fs == 16000)
        end_index[g] += 32;
    else
        end_index[g] += 64;
    if (end_index[g] > last_index[g])
        end_index[g] = last_index[g];
    layer_end_index[layer] = end_index[g];
    layer_start_cband[layer] = end_cband[g];
    end_cband[g] = layer_end_cband[layer] = (end_index[g] + 31) / 32;
}
```

where, fs is the sampling frequency.

- how to determine *layer_end_sfb[]*, the end scalefactor band to be added newly in each scalability layer
- how to determine *layer_start_sfb[]*, the start scalefactor band to be added newly in each scalability layer

```
for (g = 0; g < num_window_groups; g++)
    end_sfb[g] = 0;
    for (layer = 0; layer < (top_layer+slayer_size); layer++) {
        g = layer_group[layer];
        layer_start_sfb[layer] = end_sfb[g];
        layer_end_sfb[layer] = max_sfb;
        for (sfb = 0; sfb < max_sfb; sfb++) {
            if (layer_end_index[layer] <=
                    swb_offset_short_window[fs_index][sfb] * window_group_length[g]) {
                layer_end_sfb[layer] = sfb + 1;
                break;
            }
        }
    end_sfb[g] = layer_end_sfb[layer];
}
```

- how to determine *available_len[i]*, the available maximum size of the bitstream of the *i*-th layer. If the arithmetic coding was initialized at the beginning of the layer, 1 should subtracted from available_len[i] since the additional 1 bit is required at the arithmetic coding termination. The maximum length of the 0^{th} coding band side information(*max_cband0_si_len*) is defined as 11.

```
for (layer = 0; layer <(top_layer+slayer_size); layer++) {
    layer_si_maxlen[layer] = 0;
    for (cband = layer_start_cband[layer]; cband < layer_end_cband[layer]; cband++) {
        for (ch=0; ch <nch; ch++) {
            if (cband == 0)
                layer_si_maxlen[layer] += max_cband0_si_len;
            else
                layer_si_maxlen[layer] +=  max_cband_si_len[cband_si_type[ch]];
        }
    }
    for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++)
        for (ch = 0; ch < nch; ch++)
            layer_si_maxlen[layer] +=  max_sfb_si_len[ch] + 5;
}

for (layer = slayer_size; layer <= (top_layer + slayer_size); layer++) {
layer_bitrate = nch * ( (layer-slayer_size) * 1000 + 16000);
    layer_bit_offset[layer] = layer_bitrate * BLOCK_SIZE_SAMPLES_IN_FRAME;
    layer_bit_offset[layer] = (int)(layer_bit_offset[layer] / SAMPLING_FREQUENCY / 8 ) * 8;
    if (layer_bit_offset[layer] > frame_length*8)
        layer_bit_offset[layer] = frame_length*8;
}

for (layer = (top_layer + slayer_size –1);  layer >= slayer_size; layer--)  {
    bit_offset = layer_bit_offset[layer+1] – layer_si_maxlen[layer]
    if ( bit_offset < layer_bit_offset[layer] )
```

```
        layer_bit_offset[layer] = bit_offset
}

for (layer = slayer_size – 1; layer >= 0; layer--)
    layer_bit_offset[layer] = layer_bit_offset[layer+1] – layer_si_maxlen[layer];
overflow_size = (header_length + 7) * 8 – layer_bit_offset[0];
layer_bit_offset[0] = (header_length + 7) * 8;
if (overflow_size > 0) {
    for (layer = (top_layer+slayer_size-1); layer >= slayer_size; layer--) {
        layer_bit_size = layer_bit_offset[layer+1] – layer_bit_offset[layer];
        layer_bit_size -= layer_si_maxlen[layer];
        if (layer_bit_size >= overflow_size) {
            layer_bit_size = overflow_size;
            overflow_size = 0;
        }
        else
            overflow_size = overflow_size – layer_bit_size;
        for (m = 1; m <= layer; m++)
            layer_bit_offset[m] += layer_bit_size;
        if (overflow_size <= 0)
            break;
    }
}
else {
    underflow_size = -overflow_size;
    for (m = 1; m < slayer_size; m++)  {
        layer_bit_offset[m] = layer_bit_offset[m-1] + layer_si_maxlen[m-1];
        layer_bit_offset[m] += underflow_size / slayer_size;
        if (layer <= (underflow_size%slayer_size);
            layer_bit_offset[m] += 1;
    }
}
for (layer = 0; layer < (top_layer+slayer_size); layer++)
    available_len[layer] = layer_bit_offset[layer+1] – layer_bit_offset[layer];
```

Some help variables and arrays are needed to describe the bit-sliced decoding process of the spectral values in each BSAC fine grain layer. cur_snf[ch][g][*i*] is initialized as the MSB plane (MSBplane[ch][g][cband]) allocated to the coding band *cband,* where we can get MSBplane[][] from **cband_si[ch][g][cband]** using Table 4.A.31. And, we start the decoding of the bit-sliced data in each layer from the maximum significance, *maxsnf*.

These help variables and arrays must be built up for each bsac_spectral_data(). The pseudo code shown below describes

- how to initialize *cur_snf[][][]*, the current significance of the spectra to be added newly due to the spectral band extension in each enhancement scalability layer.

```
/* set current snf */
g = layer_group[layer];
for (ch = 0; ch < nch; ch++) {
    for (i = layer_start_index[layer]; i<layer_end_index[layer]; i++) {
        cband = i/32;
        cur_snf[ch][g][i] = MSBplane[ch][g][cband];
    }
}
```

- how to determine *maxsnf*, the maximum significance of all vectors to be decoded.

```
maxsnf = 0;
for (g = start_g; g < end_g; g++)
for (ch = 0; ch < nch; ch++) {
    for (i = start_index[g]; i< end_index[g]; i++)
        if (maxsnf < cur_snf[ch][g][i])
            maxsnf = cur_snf[ch][g][i];
}
```

- how to store *cur_snf[][][]* for the secondary coding (bsac_lower_spectra() and bsac_higher_spectra()) after the sliced bits of the new spectra has been coded in each layer (bsac_layer_spectra()).

```
/* store current snf */
for (g = 0; g < no_window_groups;g++)
for (ch = 0; ch < nch; ch++) {
   for (i = layer_start_index[layer]; i < layer_end_index[layer]; i++) {
      unc_snf[ch][g][i] = cur_snf[ch][g][i];
   }
}
```

4.5.2.6.2.6 Order of spectral coefficients in spectral_data

For ONLY_LONG_SEQUENCE windows (num_window_groups = 1, window_group_length[0] = 1) the spectral data is in ascending spectral order, as shown in Figure 4.18.

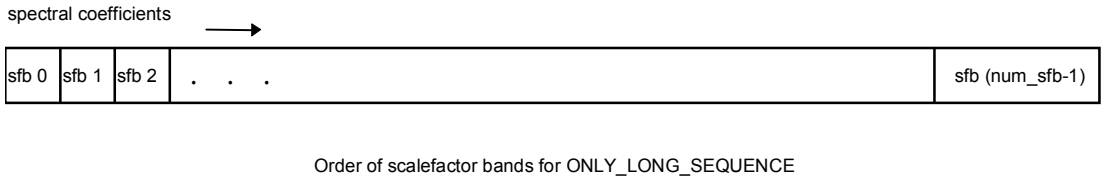


Figure 4.18 – Order of scalefactor bands for ONLY_LONG_SEQUENCE

For the EIGHT_SHORT_SEQUENCE window, each 4 spectral coefficients of blocks within each group are interleaved in ascending spectral order and the interleaved spectral coefficients are interleaved in ascending group number, as shown in Figure 4.19.

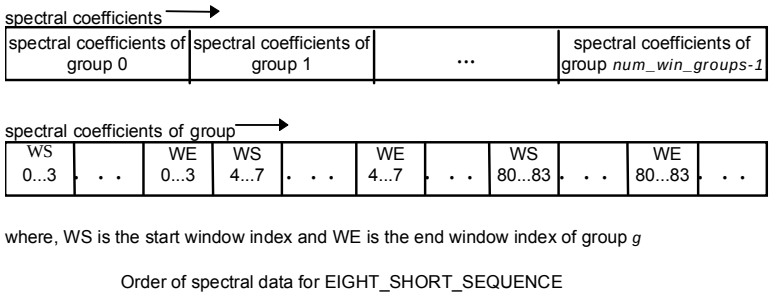


Figure 4.19 – Order of spectral data for EIGHT_SHORT_SEQUENCE

4.5.2.6.2.7 Arithmetic coding procedure

Arithmetic Coding consists of the following 2 steps:

- Initialization which is performed prior to the coding of the first symbol
- Coding of the symbol themselves.

4.5.2.6.2.7.1 Registers, symbols and constants

Several registers, symbols and constants are defined to describe the arithmetic decoder.

- half[] : 32-bit fixed point array equal to ½
- range: 32-bit fixed point register. Contains the range of the interval.
- value: 32-bit fixed point register. Contains the value of the arithmetic code.
- est_cw_len: 16-bit fixed point register. Contains the estimated length of the arithmetic codeword to be decoded.
- p0: 16-bit fixed point register (Upper 6 MSBs are available, Other LSBs are 0). Probability of the “0” symbol.

- p1: 16-bit fixed point register(Upper 6 MSBs are available, Other LSBs are 0). Probability of the “1” symbol.
- cum_freq : 16-bit fixed point registers. Cummulative Probabilities of the symbols.

4.5.2.6.2.7.2 Initialization

The bitstreams of each segment are read in the buffer of each segment. And 32-bit zero is concatenated to the buffer of each segment. If the segmented arithmetic coding is not, all the bitstreams of a frame is a segment and the zero stuffing is used. See subclause 4.6.4.6.3 for the detailed description of the segment.

The register *value* is set to 0, *range* to 1 and *est_cw_len* to 30. Using these initialized regiters, the 30 bits are read in register *value* and registers are updated when the first symbol is decoded.

4.5.2.6.2.7.3 Decoding a symbol

Arithmetic decoding procedure varies on the symbol to be decoded. If the symbol is the sliced bit of the spectral data, the binary arithmetic decoding is used. Otherwise, the general arithmetic decoding is used.

When a symbol is binary arithmetic-decoded, the probability p0 of the “0” symbol is provided according to the context computed properly and using the probability table. p0 uses a 6-bit fixed-point number representation. Since the decoder is binary, the probability of the “1” symbol is defined to be 1 minus the probability of the “0” symbol, i.e. p1 = 1-p0.

When a symbol is arithmetic-decoded, the cummulative probability values of multiple symbols are provided. The probability values are regarded as the arithmetic model. The arithmetic model for decoding a symbol is given in the data elements. For example, arithmetic models of scalefactor and cband_si are given in the data elements, **base_scf_model**, **enh_scf_model** and **cband_si_type**. Each value of the arithmetic model uses a 14-bit fixed-point representation.

4.5.2.6.2.7.4 Software

```
unsigned long half[16] =
{
    0x20000000, 0x10000000, 0x08000000, 0x04000000,
    0x02000000, 0x01000000, 0x00800000, 0x00400000,
    0x00200000, 0x00100000, 0x00080000, 0x00040000,
    0x00020000, 0x00010000, 0x00008000, 0x00004000
};
/* Initialize the Parameteres of the Arithmetic Decoder */
void initArDecode()
{
   value = 0;
   range = 1;
   est_cw_len = 30;
}
/* GENEARL ARITHMETIC DECODE */
int decode_symbol (buf_idx, cum_freq, symbol)
int buf_idx;     /* buffer index to save the arithmetic code word */
int cum_freq[];  /* Cumulative symbol freqencies */
int *symbol;     /* Symbol decoded */
{
   if (est_cw_len) {
      range = (range << est_cw_len);
      value = (value << est_cw_len) | readBits(buf_idx, est_cw_len);
         /* read bitstream from the buffer */
   }

   range >>= 14;
   cum = value/range;          /* Find cum freq */

   /* Find symbol */
   for (sym = 0; cum_freq[sym]>cum; sym++);
   *symbol = sym;

   /* Narrow the code region to that allotted to this symbol. */
   value -= (range * cum_freq[sym]);

   if (sym > 0) {
```

```
      range = range * (cum_freq[sym-1]-cum_freq[sym]);
   }
   else {
      range = range * (16384-cum_freq[sym]);
   }

   for (est_cw_len = 0; range < half[est_cw_len]; est_cw_len++) ;
   return est_cw_len;
}
/* BINARY ARITHMETIC-DECODE THE NEXT SYMBOL. */
int decode_symbol2 (buf_idx, freq0, symbol)
int buf_idx; /* buffer index to save the arithmetic code word */
int p0;      /* Normalized probability of symbol 0 */
int *symbol; /* Symbol decoded */
{
   if (est_cw_len) {
      range = (range << est_cw_len);
      value = (value << est_cw_len) | readBits(buf_idx, est_cw_len);
         /* read bitstream from the buffer */
   }

   range >>= 14;

   /* Find symbol */
   if ( (p0 * range) <= value ) {
      *symbol = 1;

      /* Narrow the code region to that allotted to this symbol. */
      value -= range * p0;
      p1 = 16384 – p0;
      range =  range * p1;
   }
   else {
      *symbol = 0;

      /* Narrow the code region to that allotted to this symbol. */
      range = range * p0;
   }

   for (est_cw_len = 0; range < half[est_cw_len]; est_cw_len++);
   return est_cw_len;
}
```

4.5.2.6.3 Error sensitivity category assignment

BSAC has the layered structure where the syntax is arranged in order of importance in order to support the fine grain scalability and error resilience. Therefore the BSAC syntax can be channel coded effectively without the bitstream reordering for advanced channel coding techniques like unequal error protection (UEP) since the error resilient syntax are included in the subclause 4.4.2.6. However, error sensitivity categories (ESC) of the data elements should be defined for advanced channel coding. The data element can be classfied into the error sensitivity categories depending upon its error sensitivity as follows:

Table 4.85 – BSAC error sensitivity category assignment

Category	data elements	description
0	frame_length, bsac_header() and general_header()	commonly used side information
1	bsac_layer_element(0)	BSAC base layer except common side
2	bsac_layer_element(1) –	1st quarternary enhancement layers
3	bsac_layer_element(top_layer/4+1) –	2nd quarternary enhancement layers
4	bsac_layer_element(top_layer/2+1) ~	3rd quarternary enhancement layers
5	basc_layer_element(top_layer*3/4+1) ~	4th quarternary enhancement layers

The lower category indicates the class with the higher error sensitivity, whereas the higher category indicates the class with the lower sensitivity.

4.5.2.7 Dynamic Range Control (DRC)

4.5.2.7.1 Definitions

pce_tag_present One bit indicating that program element tag is present;

pce_instance_tag Tag field that indicates with which program the dynamic range information is associated

drc_tag_reserved_bits Reserved

excluded_chns_present One bit indicating that excluded channels are present

drc_bands_present One bit indicating that DRC multi-band information is present

drc_band_incr Number of DRC bands greater than 1 having DRC information

drc_interpolation_scheme Indicates which interpolation scheme is used for the DRC data in the SBR QMF domain according to:

Table 4.86 – drc_interploation_scheme

drc_interpolation_scheme	Meaning
0	default interpolation
1	steep slope interpolation at position 0
2	steep slope interpolation at position 1
3	steep slope interpolation at position 2
4	steep slope interpolation at position 3
5	steep slope interpolation at position 4
6	steep slope interpolation at position 5
7	steep slope interpolation at position 6
8	steep slope interpolation at position 7
9-15	reserved

drc_band_top[i] Indicates top of i-th DRC band in units of 4 spectral lines
If drc_band_top[i]=k, then the index (w.r.t zero) of the highest spectral coefficient that is in the i-th DRC band is = k*41+3. In case of an EIGHT_SHORT_SEQUENCE window_sequence the index is interpreted as pointing into the concatenated array of 8*128 (de-interleaved) frequency points corresponding to the 8 short transforms.

prog_ref_level_present One bit indicating that reference level is present

prog_ref_level Reference level. A measure of long-term program audio level for all channels combined.

prog_ref_level_reserved_bits Reserved

dyn_rng_sgn[i] Dynamic range control sign information. One bit indicating the sign of dyn_rng_ctl (0 if positive, 1 if negative)

dyn_rng_ctl[i] Dynamic range control magnitude information

exclude_mask[i] Boolean array indicating the audio channels of a program that are excluded from DRC processing using this DRC information.

additional_excluded_chns[i] One bit indicating that additional excluded channels are present

4.5.2.7.2 Decoding process

The transport of the DRC information does not involve the MPEG-4 System layer but is handled completely within the GA data elements instead. Furthermore, the evaluation of potentially available dynamic range control information in the GA decoder is optional. No DRC related information is passed on to a subsequent audio compositor.

Advisory: Interactions between DRC and the audio compositor (see ISO/IEC 14496-1) are possible. If a mix between several audio objects in the audio compositor is intended, DRC has to be applied with care.

prog_ref_level_present indicates that **prog_ref_level** is being transmitted. This permits **prog_ref_level** to be sent as infrequently as desired (e.g. once), although periodic transmission would permit break-in.

prog_ref_level is quantized in 0.25 dB steps using 7 bits, and therefore has a range of approximately 32 dB. It indicates program level relative to full scale (i.e. dB below full scale), and is reconstructed as:

$$level = 32767 \cdot 2^{\frac{-prog\_ref\_level}{24}}$$

where „full scale level„ is 32767 (prog_ref_level equal to 0).

pce_tag_present indicates that **pce_instance_tag** is being transmitted. This permits **pce_instance_tag** to be sent as infrequently as desired (e.g. once), although periodic transmission would permit break-in.

pce_instance_tag indicates with which program the dynamic range information is associated. If this is not present then the default program is indicated. Since each AAC bitstream payload typically has just one program, this would be the most common mode. Each program in a multi-program bitstream payload would send its dynamic range information in a distinct extension_payload() of the fill_element(). In the multiple program case, the **pce_instance_tag** would always have to be signaled.

The **drc_tag_reserved_bits** fill out the optional fields to an integral number of bytes in length.

The **excluded_chns_present** bit indicates that channels that are to be *excluded* from dynamic range processing will be signaled immediately following this bit. The excluded channel mask information must be transmitted in each frame where channels are excluded. The following ordering principles are used to assign the exclude_mask to channel outputs:

- If a PCE is present, the **exclude_mask** bits correspond to the audio channels in the SCE, CPE, CCE and LFE syntax elements in the order of their appearance in the PCE. In the case of a CPE, the first transmitted mask bit corresponds to the first channel in the CPE, the second transmitted mask bit to the second channel. In the case of a CCE, a mask bit is transmitted only if the coupling channel is specified to be an independently switched coupling channel.
- If no PCE is present, the **exclude_mask** bits correspond to the audio channels in the SCE, CPE and LFE syntax elements in the order of their appearance in the bitstream payload, followed by the audio channels in the CCE syntax elements in the order of their appearance in the bitstream payload. In the case of a CPE, the first transmitted mask bit corresponds to the first channel in the CPE, the second transmitted mask bit to the second channel. In the case of CCE, a mask bit is transmitted only if the coupling channel is specified to be an independently switched coupling channel.

drc_band_incr is the number of bands greater than one if there is multi-band DRC information.

dyn_rng_ctl is quantized in 0.25 dB steps using a 7-bit unsigned integer, and therefore, in association with **dyn_rng_sgn,** has a range of +/-31.75 dB. It is interpreted as a gain value that shall be applied to the decoded audio output samples of the current frame.

The range supported by the dynamic range information is summarized in the following table:

Table 4.87 – Dynamic range information

Field	bits	steps	stepsize, dB	range, dB
prog_ref_level	7	128	0.25	31.75
dyn_rng_sgn and **dyn_rng_ctl**	1 and 7	+/- 127	0.25	+/- 31.75

The dynamic range control process is applied to the spectral data spec[i] of one frame immediately before the synthesis filterbank. In case of an EIGHT_SHORT_SEQUENCE window_sequence the index i is interpreted as pointing into the concatenated array of 8*128 (de-interleaved) frequency points corresponding to the 8 short transforms.

The following pseudo code is for illustrative purposes only, showing one method for applying one set of dynamic range control information to a frame of a target audio channel. The constants ctrl1 and ctrl2 are compression constants (typically between 0 and 1, zero meaning no compression) that may optionally be used to scale the dynamic range compression characteristics for levels greater than or less than the program reference level, respectively. The constant target_level describes the output level desired by the user, expressed in the same scaling as prog_ref_level.

```
#define FRAME_SIZE 1024 /* Change to 960 for 960-framing*/
bottom = 0;
drc_num_bands = 1;
if (drc_bands_present)
     drc_num_bands += drc_band_incr;
else
     drc_band_top[0] = FRAME_SIZE/4 - 1;
for (bd = 0; bd < drc_num_bands; bd++)  {
   top = 4 * (drc_band_top[bd] + 1);

   /* Decode DRC gain factor */
   if (dyn_rng_sgn[bd])
      factor = 2^(-ctrl1*dyn_rng_ctl[bd]/24); /* compress */
   else
      factor = 2^(ctrl2*dyn_rng_ctl[bd]/24);  /* boost */

   /* If program reference normalization is done in the digital domain, modify
    * factor to perform normalization.
    *  prog_ref_level can alternatively be passed to the system for modification
    *  of the level in the analog domain. Analog level modification avoids problems
    *  with reduced DAC SNR (if signal is attenuated) or clipping (if signal is boosted)
    */
   factor *= 0.5^((target_level-prog_ref_level)/24);

   /* Apply gain factor */
   for (i = bottom; i < top; i++)
      spec[i] *= factor;
   bottom = top;
}
```

Note the relation between dynamic range control and coupling channels:

- Dependently switched coupling channels are always coupled onto their target channels as spectral coefficients prior to the DRC and synthesis filtering of these channels. Therfore a dependently switched coupling channel's signal that couples onto to a specific target channel will undergo the DRC processing of that target channel.
- Since independently switched coupling channels couple to their target channels in the time domain, each independently switched coupling channel will undergo DRC and subsequent synthesis filtering separate from its target channels. This permits the independently switched coupling channel to have distinct DRC processing if desired.

4.5.2.7.3 Persistence of DRC information

At the beginning of a stream, all DRC information for all channels is assumed to be set to its default value: program reference level equal to the decoder's target reference level, one DRC band, with no DRC gain modification for that band. Unless this data is specifically overwritten, this remains in effect.

There are two cases for the persistence of DRC information that has been transmitted:

- The program reference level is per audio program, and persists until a new value is transmitted, at which point the new data overwrites the old and takes effect that frame. (It may be appropriate to send this value periodically to allow break-in in the case of streaming.)
- Other DRC information persists on a per-channel basis. Note that if a channel is excluded via the appropriate **exclude_mask[]** bit, then effectively no information is transmitted for that channel in that call to dynamic_range_info(). The excluded channel mask information must be transmitted in each frame where channels are excluded.

The rules for retaining per-channel DRC information are as follows:

- If there is no DRC information in a given frame for a given channel, use the information that was used in the previous frame. (This means that one adjustment can hold for a long time, although it may be appropriate to transmit the DRC information periodically to permit break-in.)
- If any DRC information for this channel appears in the current frame, the following sequence occurs: first, overwrite all per-channel DRC information for that channel with the default values (one DRC band, with no DRC gain modification for that band), then overwrite any per-channel DRC information with the transmitted values.

4.5.2.7.4 Usage of DRC with the audio object type AAC scalable

If DRC is used with the AAC scalable audio object type, the following additional restrictions and information apply:

1. The field **pce_tag_present** must be '0' (no reference to a PCE).
2. The field **excluded_chns_present** must be '0' (common control of all audio channels).
3. DRC information may be transmitted in several layers of the scalable audio object. The DRC information to be used for the DRC processing is the one carried in the highest layer available to the decoder.

4.5.2.7.5 Usage of DRC with the audio object type SBR

If DRC is used with the SBR audio object type, the process shall be applied to the spectral data in the SBR QMF domain. It is mandatory for a High Efficiency AAC Profile decoder to be able to parse the DRC extension element. The ability to decode and apply DRC data is optional for a High Efficiency AAC Profile decoder. However, if it is implemented, the implementation outlined here shall be used. The DRC for High Efficiency AAC (SBR) is fully backwards compatible with DRC for AAC without SBR.

The following pseudo code and equations show how the DRC factors are stored for use in the SBR QMF domain. The borders of the DRC bands are quantized to match the frequency resolution of the SBR QMF filterbank. In order to ensure proper backwards compatibility, the delay between the MDCT synthesis and the QMF synthesis must be considered. Hence, the DRC parameters applied to the SBR QMF subsamples shall be delayed by the same amount of time as the signal between the MDCT synthesis and QMF synthesis.

The DRC factors are stored in the matrix factorQMF[l][k], where 'l' indicates which QMF subsample the values correspond to. Since the DRC band borders relate to 1024 MDCT lines (or 960 MDCT lines for 960-framing), the borders are mapped to corresponding borders in the 32 subbands of the lower part of the SBR QMF.

For short AAC window sequences no interpolation of the DRC factors is used. For other AAC window sequences the DRC factors are interpolated over time in order to avoid zipper noise. For the DRC used with the High Efficiency AAC decoder, it is possible to signal the temporal border between the DRC factors. Hence, it is possible to control transient behaviour of the DRC without having to rely on short AAC window sequences. This is accomplished with the drc_interpolation_scheme data element.

The bands covered by the DRC data covers only the frequency range of the AAC MDCT, i.e., up to half of the sampling frequency of the AAC. The DRC data for the frequency range above that up to half of the sampling frequency of the output signal, is the same as the highest transmitted DRC band.

In the pseudo code below the differences compared to DRC application without SBR are marked by bold letters.

```
#define FRAME_SIZE 1024 /* Change to 960 for 960-framing.*/
#define NUM_QMF_SUBSAMPLES (FRAME_SIZE/32)
#define NUM_QMF_SUBSAMPLES_2 (FRAME_SIZE/64)

#if 1 /* 1024 FRAMING */
static float offset[8] = {0, 4, 8, 12, 16, 20, 24, 28};
#else /* 960 FRAMING */
static float offset[8] = {0, 4, 8, 11, 15, 19, 23, 26};
#endif

for (i =  0 ; i < 64; i++){
   for (j = 0; j < NUM_QMF_SUBSAMPLES; j++){
      factorQMF[j][i] = factorQMF[NUM_QMF_SUBSAMPLES + j][i];
   }
   previousFactors[i] = factorQMF[2*NUM_QMF_SUBSAMPLES-1][i];
}

bottom = 0;
drc_num_bands = 1;
if (drc_bands_present)
     drc_num_bands += drc_band_incr;
if (!drc_bands_present)
     drc_band_top[0] = FRAME_SIZE/4 - 1;
for (bd = 0; bd < drc_num_bands; bd++)  {
   top    = 4 * (drc_band_top[bd] + 1);

   /* Decode DRC gain factor */
   if (dyn_rng_sgn[bd])
      factor = 2^(-ctrl1*dyn_rng_ctl[bd]/24); /* compress */
```

```
else
    factor = 2^(ctrl2*dyn_rng_ctl[bd]/24);  /* boost */

/* If program reference normalization is done in the digital domain, modify
 * factor to perform normalization.
 *  prog_ref_level can alternatively be passed to the system for modification
 *  of the level in the analog domain. Analog level modification avoids problems
 *  with reduced DAC SNR (if signal is attenuated) or clipping (if signal is boosted)
 */
factor *= 0.5^((target_level-prog_ref_level)/24);

/* Truncate bottom and top to be a multiple of NUM_QMF_SUBSAMPLES.*/
if(bT != EIGHT_SHORT_WINDOW){
    bottom = NUM_QMF_SUBSAMPLES*(int)(bottom/NUM_QMF_SUBSAMPLES);
    top    = NUM_QMF_SUBSAMPLES*(int)(top/NUM_QMF_SUBSAMPLES);
}
else{
    bottom = (int)(NUM_QMF_SUBSAMPLES/8*(int)(bottom*8/NUM_QMF_SUBSAMPLES));
    top    = (int)(NUM_QMF_SUBSAMPLES/8*(int)(top*8/NUM_QMF_SUBSAMPLES));
}

/* Apply gain factor */
/* for (i = bottom; i < top; i++)
       spec[i] *= factor;
*/

if(bT != EIGHT_SHORT_WINDOW){ /* bt indicates the AAC window sequence for the
                                 current frame. */
    bottomQMF = bottom*32/FRAME_SIZE;
    for (j = -NUM_QMF_SUBSAMPLES_2; j < NUM_QMF_SUBSAMPLES; j++){
        alphaValue = 0;
        if(j+NUM_QMF_SUBSAMPLES_2 < NUM_QMF_SUBSAMPLES){
            if(p->drc_interp_scheme == 0){
                k = 1.0f / ((float) NUM_QMF_SUBSAMPLES);
                alphaValue = (j+NUM_QMF_SUBSAMPLES_2)*k;
            }
            else{
                if (j+NUM_QMF_SUBSAMPLES_2 >= offset[p->drc_interp_scheme - 1])
                    alphaValue = 1;
            }
        }
        else
            alphaValue = 1;
        for (i = bottomQMF ; i < 64 ; i++) {
            factorQMF[NUM_QMF_SUBSAMPLES + j ][i] = alphaValue*factor +
                                (1 - alphaValue)*previousFactors[i];
        }
    }
}
else{
 /*   StartSample and stopSample indicate the borders in QMF subsamples for the current DRC
      data. They are derived by first removing all integer multiples of the short window
      length. Since there are always 32 QMF analysis subbands for FRAME_SIZE input frame
      size, the number of QMF subsamples is FRAME_SIZE/32 (NUM_QMF_SUBSAMPLES) for every
      frame. Hence the number of QMF subsamples for a short window is
      (NUM_QMF_SUBSAMPLES)/8. bottomQMF is calculated from the remainder when the integer
      multiples of the short window length have been removed. This index to an MDCT line is
      mapped to the corresponding subband in the QMF by multiplying by 32/(FRAME_SIZE/8).
      If the the borders between two short windows are crossed, i.e. startSample is located
      in short window n-1 and stopSample is located in short window n, the bottomQMF needs
      to be set to zero when the border between the frames is crossed.
*/

    /* startSample is truncated to the corresponding start subsample in
the QMF of the short window bottom is present in.*/
    startSample = floor((float)
                            bottom/(FRAME_SIZE/8.0f))*(NUM_QMF_SUBSAMPLES)/8;

    /* stopSample is rounded upwards to the nearest corresponding stop subsample
        in the QMF of the short window top is present in.*/
```

```
        stopSample = ceil ((float) top/(FRAME_SIZE/8.0f))*(NUM_QMF_SUBSAMPLES)/8;
        bottomQMF  = (bottom%(FRAME_SIZE/8))*32/(FRAME_SIZE/8);
        for (j = startSample; j < stopSample; j++){
            if(j > startSample && j%4 == 0){
                bottomQMF = 0;
            }
            for (i = bottomQMF ; i < 64 ; i++) {
                factorQMF[NUM_QMF_SUBSAMPLES + j][i] = factor;
            }
        }
    }
    bottom = top;
}
```

The different interpolation windows calculated based on drc_interpolation_scheme in the pseudo code above are displayed in the following figure. For all drc_interpolation_scheme values except zero, a step function is used. For the first frame that is not an eight short AAC window sequence frame, following an eight short AAC window sequence, the drc_interpolation_sheme should be five. In the lower half of the figure an example of a window sequence is given, where the line corresponding to one minus the window value is also drawn (labeled 1-alphaValue).

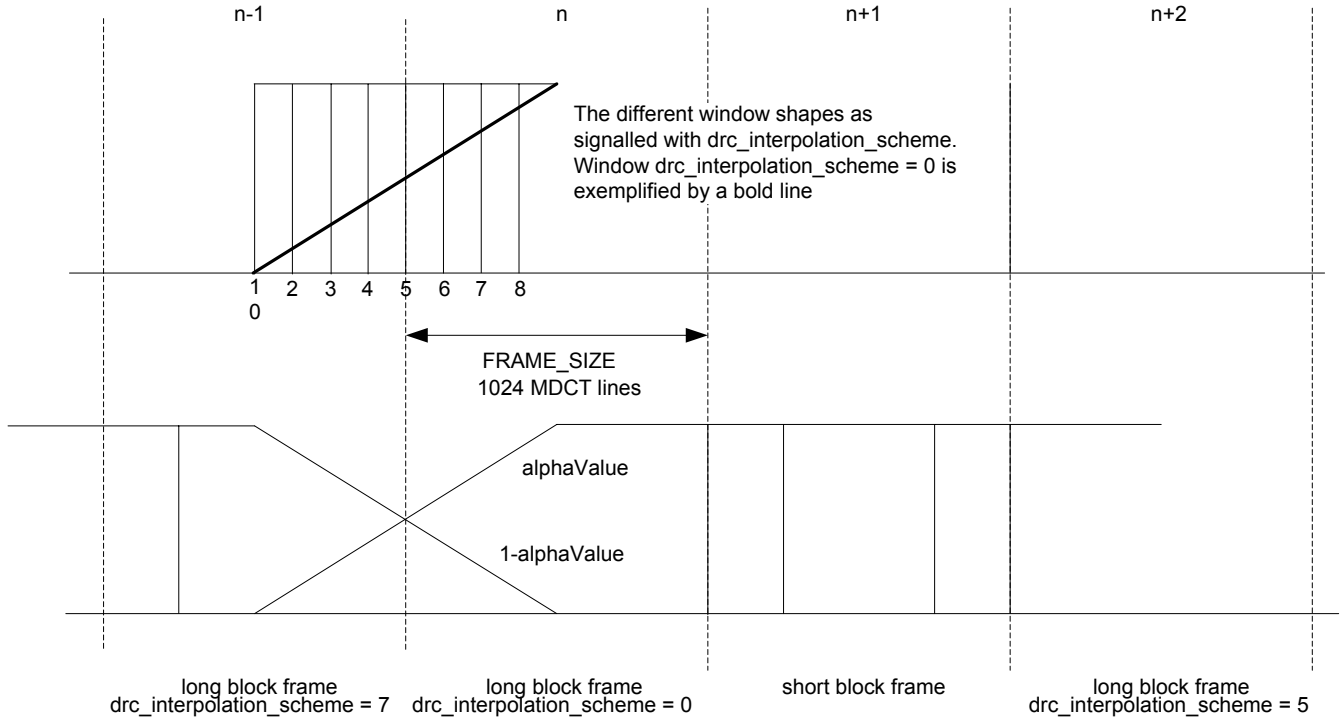


Figure 4.20 – Interpolation windows and window sequence example

The DRC is applied immediately prior to the SBR QMF synthesis by multiplying every element in the matrix holding the subbands samples to be synthesised by the corresponding element in *factor(k,l)*, where $factor(k,l) = \texttt{factorQMF}[RATE \cdot numTimeSlots - l_{border} + l][k]$, and l_{border} is calculated as follows

$$l_{border} = \frac{\frac{FRAME\_SIZE}{2} + \frac{L_{Analysisfilter}}{2} - N_{analysisChannles} + Delay_{buffer}}{N_{analysisChannles}} = \begin{cases} 26 & for\ FRAME\_SIZE = 1024 \\ 25 & for\ FRAME\_SIZE = 960 \end{cases}$$

where

$$N_{analysisChannles} = 32\,,\ L_{Analysisfilter} = 320 \text{ and } Delay_{buffer} = 6 \cdot 32\,.$$

4.5.2.8 Payloads for the audio object type SBR

4.5.2.8.1 Definitions

bs_reserved Bits reserved for future use, default value is zero.

sbr_extension_data() Syntactic element that contains the SBR extension data.

bs_sbr_crc_bits Cyclic redundancy checksum for the SBR extension data. The CRC code is defined by the generator polynomial $G_{10}(x) = x^{10} + x^9 + x^5 + x^4 + x + 1$ and the initial value for the CRC calculation is zero.

bs_header_flag Indicates if an SBR header is present.

sbr_header() Syntactic element that contains the SBR header.

sbr_data() Syntactic element that contains the SBR data.

bs_fill_bits Byte alignment bits.

bs_amp_res Defines the resolution of the envelope estimates according to:

Table 4.88 – bs_amp_res

bs_amp_res	Meaning
0	1.5 dB
1	3.0 dB

bs_start_freq Input parameter to function that calculates start of master frequency table.

bs_stop_freq Input parameter to function that calculates stop of master frequency table.

bs_xover_band Index to master frequency table.

bs_header_extra_1 Indicates if the optional header part 1 is present.

bs_header_extra_2 Indicates if the optional header part 2 is present.

bs_freq_scale Input parameter to function that calculates master frequency table, defined by:

Table 4.89 – bs_freq_scale

bs_freq_scale	Meaning
0	linear
1	12 bands/octave
2 (default)	10 bands/octave
3	8 bands/octave

bs_alter_scale Input parameter to function that calculates master frequency table, defined by:

Table 4.90 – bs_alter_scale

bs_alter_scale	**Meaning** for bs_freq_scale = 0	**Meaning** for bs_freq_scale > 0
0	no grouping of channels	no alteration
1 (default)	groups of 2 channels	extra wide bands in highest range

bs_noise_bands Input parameter to function that calculates noise band table, defined by:

Table 4.91 – bs_noise_bands

bs_noise_bands	Meaning
0	1 band
1	1 band/octave
2 (default)	2 bands/octave
3	3 bands/octave

bs_limiter_bands Input parameter to function that calculates limiter band table, defined by:

Table 4.92 – bs_limiter_bands

bs_limiter_bands	Meaning	Note
0	1 band	single band
1	1.2 bands/octave	multi-band
2 (default)	2.0 bands/octave	multi-band
3	3.0 bands/octave	multi-band

bs_limiter_gains Defines the maximum gain of the limiters according to:

Table 4.93 – bs_limiter_gains

bs_limiter_gains	Meaning
0	-3 dB Max Gain
1	0 dB Max Gain
2 (default)	3 dB Max Gain
3	inf. dB Max Gain(i.e. limiter off)

bs_interpol_freq Defines if the frequency interpolation shall be applied according to:

Table 4.94 – bs_interpol_freq

bs_interpol_freq	Meaning
0	off
1 (default)	on

bs_smoothing_mode Defines if smoothing shall be applied according to:

Table 4.95 – bs_smoothing_mode

bs_smoothing_mode	Meaning
0	on
1 (default)	off

sbr_single_channel_element() Syntactic element that contains data for an SBR single channel element.
sbr_channel_pair_element() Syntactic element that contains data for an SBR channel pair element.
sbr_channel_pair_base_element() Syntactic element that contains base-layer data for an SBR channel pair element, in a scalable system.
sbr_channel_pair_enhance_element() Syntactic element that contains enhancement-layer data for an SBR channel pair element, in a scalable system.
bs_data_extra Indicates if reserved bits in the SBR data part are present.
sbr_grid() Syntactic element that contains the time frequency grid.
sbr_dtdf() Syntactic element that contains the information on how the envelope and noise data is delta coded.
sbr_invf() Syntactic element that contains the inverse filtering data.
sbr_envelope() Syntactic element that contains the huffman coded envelope data.
sbr_noise() Syntactic element that contains the huffman coded noise floor data.
bs_add_harmonic_flag Indicates if sinusoidal coding information is present.
sbr_sinusoidal_coding() Syntactic element that contains sinusoidal coding data.
bs_extended_data Indicates if an SBR extended data element is present.
bs_extension_size Defines the size of the SBR extended date element in bytes.
bs_esc_count Further defines the size of the SBR extended data element in cases where the size is larger than 14 bytes.
bs_extension_id Defines the ID of the SBR extended data element, for future use, according to:

Table 4.96 – bs_extension_id

bs_extension_id	Meaning
0	reserved
1	reserved
2	reserved

3	reserved

sbr_extension() Syntactic element for future extensions.

bs_coupling Indicates if the stereo information between the two channels is coupled or not according to:

Table 4.97 – bs_coupling

bs_coupling	Meaning
0	the channels are not coupled
1	the channels are coupled

bs_frame_class Indicates the frame class of the current SBR frame according to:

Table 4.98 – bs_frame_class

bs_frame_class	Meaning
0	FIXFIX
1	FIXVAR
2	VARFIX
3	VARVAR

tmp Helper variable used in sbr_grid().

bs_num_env Indicates the number of SBR envelopes in the current SBR frame.

bs_freq_res Indicates the frequency resolution for each channel and SBR envelope according to:

Table 4.99 – bs_freq_res vector element

bs_freq_res[]	Meaning
0	low frequency resolution
1	high frequency resolution

bs_pointer Pointer to a specific border.

bs_var_bord_0 Indicates the position of the leading variable border for class VARVAR and VARFIX.

bs_var_bord_1 Indicates the position of the trailing variable border for class VARVAR and FIXVAR.

bs_num_rel_0 Indicates number of relative borders starting from bs_var_bord_0.

bs_num_rel_1 Indicates number of relative borders starting from bs_var_bord_1.

bs_rel_bord_0 Indicates the lengths of the relative borders starting from bs_var_bord_0.

bs_rel_bord_1 Indicates the lengths of the relative borders starting from bs_var_bord_1.

bs_df_env Indicates delta coding direction for each SBR envelope according to:

Table 4.100 – bs_df_env vector element

bs_df_env[]	Meaning
0	apply delta decoding in frequency direction for the indicated frequency band
1	apply delta decoding in time direction for the indicated frequency band

bs_df_noise Indicates delta coding direction for each noise floor according to:

Table 4.101 – bs_df_noise vector element

bs_df_noise[]	Meaning
0	apply delta decoding in frequency direction for the indicated frequency band
1	apply delta decoding in time direction for the indicated frequency band

bs_invf_mode Indicates the level of inverse filtering for each frequency band according to:

Table 4.102 – bs_invf_mode vector element:

bs_invf_mode[]	Meaning
0	no inverse filtering
1	low level inverse filtering
2	intermediate inverse filtering
3	strong inverse filtering

bs_env_start_value_balance Holds the first envelope scalefactor in case of a coupled stereo bitstream payload.

bs_data_env Holds the raw envelope scalefactors for each channel, SBR envelope and band.

bs_env_start_value_level Holds the first envelope scalefactor in case of a non-coupled stereo or a mono bitstream payload.

sbr_huff_dec() Huffman decoder defined in Annex 4.A.6

bs_codeword Huffman code word.

bs_noise_start_value_balance Holds the first noise floor value in case of a coupled stereo bitstream payload.

bs_data_noise Holds the raw noise floor data for each envelope and band.

bs_noise_start_value_level Holds the first noise floor value in case of a non-coupled stereo or a mono bitstream payload.

bs_add_harmonic Indicates if a sinusoidal should be added to a specific frequency band according to:

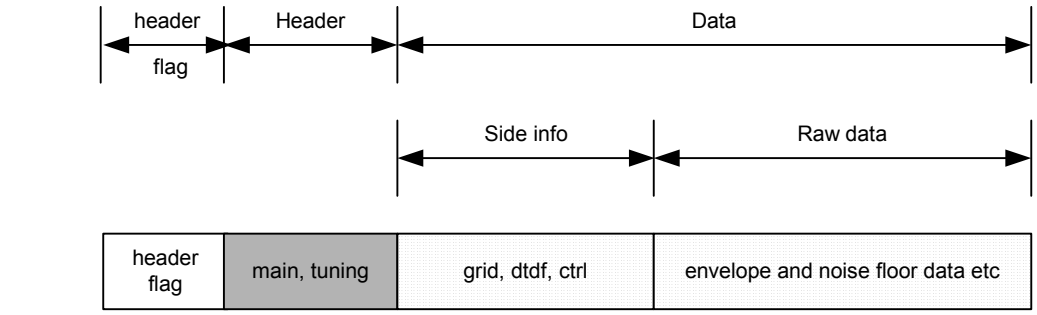
Table 4.103 – bs_add_harmonic vector element

bs_add_harmonic[]	Meaning
0	do not add a sinusoidal to the indicated frequency band
1	add a sinusoidal to the indicated frequency band

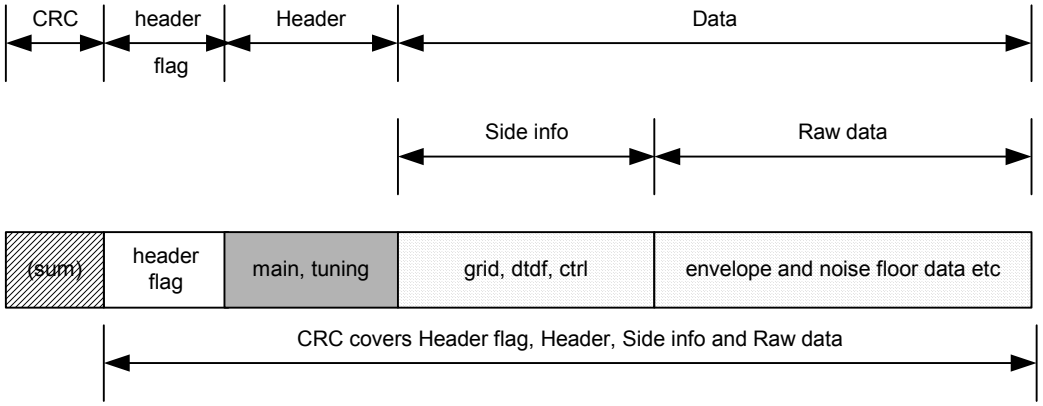
4.5.2.8.2 Decoding process

4.5.2.8.2.1 SBR frame overview

An overview of the contents of the two possible SBR extension data elements is given in Figure 4.21 below.



a) SBR extension data element



b) SBR extension data element with CRC

Figure 4.21 – The basic sections of the SBR extension data elements

The CRC field (if applicable) holds a Cyclic Redundancy Code checksum of 10 bit length. The checksum shall be calculated covering the whole SBR data range including possible bs_fill_bits.

The bs_header_flag, if set, indicates that an SBR header part is present. The SBR header part contains fundamental information such as SBR frequency range (denoted as main in Figure 4.21), as well as control signals that do not require frequent changes (denoted as tuning). Prior to SBR decoding, a SBR header part must be present. As long as no SBR header part is present, the SBR decoder performs upsampling and delay adjustment only. In continuous broadcast applications, SBR extension data elements with an SBR header part are typically sent twice per second. In addition, a SBR header part can any time be inserted, if an instantaneous, possibly program dependent, change of header parameters is required.

The SBR data part can be subdivided into side info and raw data, where side info is defined as signals needed to decode the raw data and some decoder tuning signals. Raw data consists of Huffman coded envelope scalefactors and noise floor estimates. The grid part describes how the current SBR frame is subdivided in time into time segments, and the frequency resolution of those time segments. The dtdf part signals how the data is encoded (delta coding in time or frequency direction). Channel configuration issues and decoding procedures are discussed in detail in subclause 4.6.18.3.

4.5.2.8.2.2 SBR extension payload for the audio object types AAC main, AAC SSR, AAC LC and AAC LTP

One SBR fill element is used per AAC syntactic element that is to be enhanced by SBR. SBR elements are inserted into the raw_data_block() after the corresponding AAC elements. Each AAC SCE, CPE or independently switched CCE must be succeeded by a corresponding SBR element. LFE elements are decoded according to standard AAC procedures but must be delay adjusted and re-sampled to match the output sample rate. Given

below is an example of the structure of syntactic elements within a raw data block of a 5.1 (multi-channel) configuration, where SBR is used without a CRC check.

```
<SCE> <FIL <EXT_SBR_DATA(SCE)>>      // center
<CPE> <FIL <EXT_SBR_DATA(CPE)>>      // front L/R
<CPE> <FIL <EXT_SBR_DATA(CPE)>>      // back L/R
<LFE>                                // sub
<END>                                // (end of raw data block)
```

The time domain mix of an independently switched CCE is done after SBR decoding. A dependently switched CCE is first added to the target SCE or CPE channels and SBR is applied after this addition.

4.5.2.8.2.3 SBR extension payload for the audio object types ER AAC LC and ER AAC LTP

The number and the order of the SBR extension data elements (if present) is given by the channelConfiguration. To each SCE or CPE in one er_raw_data_block(), there is a corresponding SBR extension_payload() containing either sbr_extension_data(ID_SCE) or sbr_extension_data(ID_CPE). There is no SBR extension_payload() for LFE. LFE elements are decoded according to standard AAC procedures but must be delay adjusted and re-sampled to match the output sample rate. Only SBR extension data elements without CRC check are allowed for the audio object types ER AAC LC and ER AAC LTP. The SBR extension elements shall be placed after any other extension elements. Given below is an example of the structure of syntactic elements for channelConfiguration 6.

<SCE> <CPE> <CPE> <LFE> <EXT <SBR(SCE)>> <EXT <SBR(CPE)>> <EXT <SBR(CPE)>>

4.5.2.8.2.4 SBR extension payload for the audio object types AAC scalable and ER AAC scalable

Bandwidth scalability and mono/stereo scalability of the AAC core coder is supported by the SBR bandwidth extension tool. For core coder bandwidth scalability, the SBR data is transmitted in the lowest AAC core coder layer and the SBR data covers the largest SBR frequency range used in the scalable system. The start frequency transmitted for the SBR frequency range is set to the lowest upper frequency of the AAC core coder frequency range. Hence, such a core coder bandwidth scalable system with SBR provides a constant audio bandwidth for all layers. If only the lowest layer is available for decoding, the high frequency range is covered by SBR. If higher layers are available, the high frequency range is partly (or even completely) covered by AAC, and the SBR signal is only partly needed for the remaining upper part of the high frequency range that is not covered by AAC.

The scalable SBR data is embedded into the MPEG-4 stream in the same way as for non-scalable SBR data elements, by means of using the extension_payload(). The SBR extension elements shall be placed after any other extension elements. However, only the lowest layer, or, in case mono/stereo scalability is chosen, in addition also the lowest stereo layer, carries an SBR data element. The different types of scalable SBR layers for all possible configurations of bandwidth scalability and mono/stereo scalability of the AAC core coder are described in Table 4.104.

Table 4.104 – SBR data in scalable SBR layers

Description	sbr_layer	sbr_extension_data() in extension_payload()
Non-scalable AAC core coder	SBR_NOT_SCALABLE	yes
First layer of scalable AAC core coder (mono)	SBR_MONO_BASE	yes
First stereo AAC layer in a mono/stereo scalable AAC core coder configuration	SBR_STEREO_ENHANCE	yes
First layer of scalable AAC core coder (stereo)	SBR_STEREO_BASE	yes
All other layers of a scalable AAC core coder		no

4.5.2.9 Extension payload

4.5.2.9.1 Data elements

extension_type Four bit field indicating the type of fill element content, see Table 4.105.

fill_nibble Four bit field for fill, shall be set to ‘0000’.

fill_byte Byte to be discarded by the decoder, shall be set to ‘10100101’ (to ensure that self-clocked data streams, such as radio modems, can perform reliable clock recovery).

data_element_version Four bit field indicating the version of the data element, see Table 4.106.

dataElementLengthPart a field indicating the length of the extension payload ‘data element‘. The value 255 is used as an escape value and indicates that at least one more **dataElementLengthPart** value is following. The overall length of the transmitted ‘data element‘ is calculated by summing up the partial values.

data_element_byte: a variable indicating the partial values of the extension payload ‘data element‘ with type ‘ANC_DATA‘ in bytes

other_bits bits to be discarded by the decoder.

4.5.2.9.2 Helper elements

align a helper variable indicating the amount of bits which are already processed to fulfill byte alignment requirements in extension payload.

loopCounter a helper variable indicating the length of the variable dataElementLength in bytes.

dataElementLength a helper variable indicating the length of the extension payload ‘data element‘.

4.5.2.9.3 Tables

Table 4.105 – Values of the extension_type field

Symbol	Value of extension_type	Purpose
EXT_FILL	‘0000’	bitstream payload filler
EXT_FILL_DATA	‘0001’	bitstream payload data as filler
EXT_DATA_ELEMENT	’0010‘	data element
EXT_DYNAMIC_RANGE	‘1011’	dynamic range control
EXT_SBR_DATA	‘1101’	SBR enhancement
EXT_SBR_DATA_CRC	‘1110’	SBR enhancement with CRC
-	all other values	Reserved: These values can be used for a further extension of the syntax in a compatible way.
Note: Extension payloads of the type EXT_FILL or EXT_FILL_DATA have to be added to the bitstream payload if the total bits for all audio data together with all additional data are lower than the minimum allowed number of bits in this frame necessary to reach the target bitrate. Those extension payloads are avoided under normal conditions and free bits are used to fill up the bit reservoir. Those extension payloads are written only if the bit reservoir is full.		

4.5.2.9.4 Decoding process

Table 4.106 – Values of the data_element_version

Symbol	Value of data_element_version	Purpose
ANC_DATA	‘0000‘	Ancillary data element
-	all other values	Reserved

4.5.3 Buffer requirements

4.5.3.1 Minimum decoder input buffer

The following rules are used to calculate the maximum number of bits in the input buffer both for the bitstream payload as a whole, for any given program, or for any given SCE/CPE/CCE:

The input buffer size is 6144 bits per SCE or independently switched CCE, plus 12288 bits per CPE (6144*NCC). Both the total buffer and the individual buffer sizes are limited, so that the buffering limit can be calculated for either the entire bitstream payload, any entire program, or the individual audio elements permitting the decoder to break a multichannel bitstream payload into separate mono and stereo bitstream payloads which are decoded by separate mono and stereo decoders, respectively. All bits for LFE's or dependent CCE's must be supplied from the total buffer requirements based on the independent CCE's, SCE'sand CPE's. Furthermore, all bits required for any DSE's, PCE's, FIL's, or fixed headers, variable headers, byte_alignment, and CRC must also be supplied from the same total buffer requirements.

For any error protected payload a supplementary decoder input buffer is specified. It is (N+5) % larger than the input buffer for the unprotected payload as specified above, where N is the value of "max. redundancy by class FEC" as specified in the level definition of any appropriate profile.. Furthermore, all bits required for any DSE's, PCE's, FIL's, or fixed headers, variable headers, byte_alignment, and CRC must also be supplied from the same total buffer requirements.

For the audio object type AAC scalable the same restrictions apply, however, here they apply for the combined size of the input buffers of all ASME and ASEE. This means, if a mono program is encoded a buffer size of 6144=1024*6 bits is required, and for a stereo program a total buffer size of 12288 bits is available. In the case of the scalable configurations with both, mono- and stereo-layers, the maximum buffer size for all mono layers is 6144 bits. The total buffer size for all layers is 12288 bits.

4.5.3.2 Bit reservoir

The bit reservoir is controlled at the encoder. The maximum bit reservoir in the encoder depends on the NCC and the mean bitrate. The maximum bit reservoir size for constant rate channels can be calculated by subtracting the mean number of bits per block from the minimum decoder input buffer size. For example, at 96 kbit/s for a stereo signal at 44.1 kHz sampling frequency the mean number of bits per block (mean_framelength) is (96000 bit/s / 44100 1/s * 1024)=2229.1156… .This leads to a maximum bit reservoir size (max_bit_reservoir) of INT(12288 bit - 2229.1156…)=10058. For variable bitrate channels the encoder must operate in a way that the input buffer requirements do not exceed the minimum decoder input buffer.

The state of the bit reservoir (bit_reservoir_state) is transmitted in the buffer_fullness field, either as the state of the bit reservoir truncated to an integer value (adif_buffer_fullness) or as the state of the bit reservoir divided by the NCC divided by 32 and truncated to an integer value (adts_buffer_fullness).

The bit_reservoir_state of subsequent frames can be derived as follows:

$$bit\_reservoir\_state[frame] = bit\_reservoir\_state[frame-1] + mean\_framelength - framelength[frame]$$

Framelengths have to be adjusted such that the following restriction is met:

$$0 \le bit\_reservoir\_state[frame] \le max\_bit\_reservoir$$

4.5.3.3 Maximum bit rate

The maximum bitrate depends on the audio sampling rate. It can be calculated based on the minimum input buffer size according to the formula:

$$\frac{6144\frac{bit}{block}}{1024\frac{samples}{block}} \cdot sampling\_frequency \cdot NCC$$

Table 4.107 gives some examples of the maximum bitrates per channel depending on the used sampling frequency.

Table 4.107 – Maximum bitrate depending on the sampling frequency

sampling_frequency	maximum bitrate / NCC
48 kHz	288 kbit/s
44.1 kHz	264.6 kbit/s
32 kHz	192 kbit/s

4.5.4 *Tables*

Table 4.108 – Transform windows

Window	num_swb	#coeffs	looks like
LONG_WINDOW	49	1024/960	
SHORT_WINDOW	14	128/120	
LONG_START_WINDOW	49	1024/960	
LONG_STOP_WINDOW	49	1024/960	

Table 4.109 – Window Sequences

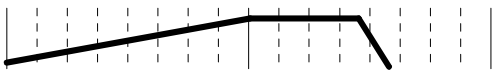
Value	window_sequence	num_ windows	looks like
0	ONLY_LONG_SEQUENCE = LONG_WINDOW	1	
1	LONG_START_SEQUENCE = LONG_START_WINDOW	1	
2	EIGHT_SHORT_SEQUENCE = 8 * SHORT_WINDOW	8	
3	LONG_STOP_SEQUENCE = LONG_STOP_WINDOW	1	

Table 4.110 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 44.1 and 48 kHz

fs [kHz]	44.1,48
num_swb_long_window	49
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	48
12	56
13	64
14	72
15	80
16	88
17	96
18	108
19	120
20	132
21	144
22	160
23	176
24	196

swb	swb_offset_long_window
25	216
26	240
27	264
28	292
29	320
30	352
31	384
32	416
33	448
34	480
35	512
36	544
37	576
38	608
39	640
40	672
41	704
42	736
43	768
44	800
45	832
46	864
47	896
48	928
	1024 (960)

Table 4.111 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 32, 44.1 and 48 kHz

fs [kHz]	32,44.1,48
num_swb_short_window	14
swb	swb_offset_short_window
0	0
1	4
2	8
3	12
4	16
5	20
6	28
7	36

swb	swb_offset_short_window
8	44
9	56
10	68
11	80
12	96
13	112
	128 (120)

Table 4.112 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 32 kHz

fs [kHz]	32
num_swb_long_window	51
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	48
12	56
13	64
14	72
15	80
16	88
17	96
18	108
19	120
20	132
21	144
22	160
23	176
24	196
25	216

swb	swb_offset_long_window
26	240
27	264
28	292
29	320
30	352
31	384
32	416
33	448
34	480
35	512
36	544
37	576
38	608
39	640
40	672
41	704
42	736
43	768
44	800
45	832
46	864
47	896
48	928
49	960
50	992 (-)
	1024 (-)

Table 4.113 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 8 kHz

fs [kHz]	8
num_swb_long_window	40
swb	swb_offset_long_window
0	0
1	12
2	24
3	36
4	48
5	60
6	72
7	84
8	96
9	108
10	120
11	132
12	144
13	156
14	172
15	188
16	204
17	220
18	236
19	252
20	268

swb	Swb_offset_long_window
21	288
22	308
23	328
24	348
25	372
26	396
27	420
28	448
29	476
30	508
31	544
32	580
33	620
34	664
35	712
36	764
37	820
38	880
39	944
	1024 (960)

Table 4.114 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 8 kHz

fs [kHz]	8
num_swb_short_window	15
swb	swb_offset_short_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28

swb	swb_offset_short_window
8	36
9	44
10	52
11	60
12	72
13	88
14	108
	128 (120)

Table 4.115 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 11.025, 12 and 16 kHz

fs [kHz]	11.025, 12, 16
num_swb_long_window	43
swb	swb_offset_long_window
0	0
1	8
2	16
3	24
4	32
5	40
6	48
7	56
8	64
9	72
10	80
11	88
12	100
13	112
14	124
15	136
16	148
17	160
18	172
19	184
20	196
21	212

swb	swb_offset_long_window
22	228
23	244
24	260
25	280
26	300
27	320
28	344
29	368
30	396
31	424
32	456
33	492
34	532
35	572
36	616
37	664
38	716
39	772
40	832
41	896
42	960
	1024 (-)

Table 4.116 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 11.025, 12 and 16 kHz

fs [kHz]	11.025, 12, 16
num_swb_short_window	15
swb	swb_offset_short_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28

swb	swb_offset_short_window
8	32
9	40
10	48
11	60
12	72
13	88
14	108
	128 (120)

Table 4.117 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 22.05 and 24 kHz

fs [kHz]	22.05 and 24
num_swb_long_window	47
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	52
13	60
14	68
15	76
16	84
17	92
18	100
19	108
20	116
21	124
22	136
23	148

swb	swb_offset_long_window
24	160
25	172
26	188
27	204
28	220
29	240
30	260
31	284
32	308
33	336
34	364
35	396
36	432
37	468
38	508
39	552
40	600
41	652
42	704
43	768
44	832
45	896
46	960
	1024 (-)

Table 4.118 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 22.05 and 24 kHz

fs [kHz]	22.05 and 24
num_swb_short_window	15
swb	swb_offset_short_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28

swb	swb_offset_short_window
8	36
9	44
10	52
11	64
12	76
13	92
14	108
	128 (120)

Table 4.119 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 64 kHz

fs [kHz]	64
num_swb_long_window	47 (46)
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	64
16	72
17	80
18	88
19	100
20	112
21	124
22	140
23	156

swb	swb_offset_long_window
24	172
25	192
26	216
27	240
28	268
29	304
30	344
31	384
32	424
33	464
34	504
35	544
36	584
37	624
38	664
39	704
40	744
41	784
42	824
43	864
44	904
45	944
46	984 (960)
	1024 (-)

Table 4.120 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 64 kHz

fs [kHz]	64
num_swb_short_window	12
swb	swb_offset_short_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24

swb	swb_offset_short_window
7	32
8	40
9	48
10	64
11	92
	128 (120)

Table 4.121 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 88.2 and 96 kHz

fs [kHz]	88.2 and 96
num_swb_long_window	41
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	64
16	72
17	80
18	88
19	96
20	108

swb	swb_offset_long_window
21	120
22	132
23	144
24	156
25	172
26	188
27	212
28	240
29	276
30	320
31	384
32	448
33	512
34	576
35	640
36	704
37	768
38	832
39	896
40	960
	1024 (-)

Table 4.122 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 88.2 and 96 kHz

fs [kHz]	88.2 and 96
num_swb_short_window	12
swb	swb_offset_short_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24

swb	swb_offset_short_window
7	32
8	40
9	48
10	64
11	92
	128 (120)

Table 4.123 – scalefactor bands for a window length of 960 at 44.1 and 48 kHz

fs [kHz]	44.1, 48
num_swb_long_window	35
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	64
16	72
17	80

swb	swb_offset_long_window
18	88
19	96
20	108
21	120
22	132
23	144
24	156
25	172
26	188
27	212
28	240
29	272
30	304
31	336
32	368
33	400
34	432
	480

Table 4.124 – scalefactor bands for a window length of 1024 at 44.1 and 48 kHz

fs [kHz]	44.1, 48
num_swb_long_ window	36
swb	swb_offset_long_ window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	60
16	68
17	76
18	84

swb	swb_offset_long_ window
19	92
20	100
21	112
22	124
23	136
24	148
25	164
26	184
27	208
28	236
29	268
30	300
31	332
32	364
33	396
34	428
35	460
	512

Table 4.125 – scalefactor bands for a window length of 960 at 32 kHz

fs [kHz]	32
num_swb_long_ window	37
Swb	swb_offset_long_ window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	60
16	64
17	72
18	80

Swb	swb_offset_long_ window
19	88
20	96
21	104
22	112
23	124
24	136
25	148
26	164
27	180
28	200
29	224
30	256
31	288
32	320
33	352
34	384
35	416
36	448
	480

Table 4.126 – scalefactor bands for a window length of 1024 at 32 kHz

fs [kHz]	32
num_swb_long_ window	37
Swb	swb_offset_long_ window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	64
16	72
17	80
18	88

swb	swb_offset_long_ window
19	96
20	108
21	120
22	132
23	144
24	160
25	176
26	192
27	212
28	236
29	260
30	288
31	320
32	352
33	384
34	416
35	448
36	480
	512

Table 4.127 – scalefactor bands for a window length of 960 at 22.05 and 24 kHz

fs [kHz]	24, 22.05
num_swb_long_ window	30
swb	swb_offset_long_ window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	52
13	60
14	68
15	80

swb	swb_offset_long_ window
16	92
17	104
18	120
19	140
20	164
21	192
22	224
23	256
24	288
25	320
26	352
27	384
28	416
29	448
	480

Table 4.128 – scalefactor bands for a window length of 1024 at 22.05 and 24 kHz

fs [kHz]	24, 22.05
num_swb_long_ window	31
swb	swb_offset_long_ window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	52
13	60
14	68
15	80

swb	swb_offset_long_ window
16	92
17	104
18	120
19	140
20	164
21	192
22	224
23	256
24	288
25	320
26	352
27	384
28	416
29	448
30	480
	512

Table 4.129 – AAC error sensitivity category assignment for main payload

SCE / LFE / mono layer	CPE, common_window == 0	CPE, common_window == 1 / stereo layer	data_element	function
1	-	0	max_sfb	aac_scalable_extension_header()
-	-	0	ms_mask_present	aac_scalable_extension_header()
1	-	1	tns_data_present	aac_scalable_extension_header()

SCE / LFE / mono layer	CPE, common_window == 0	CPE, common_window == 1 / stereo layer	data_element	function
1	-	0	ics_reserved_bit	aac_scalable_main_header()
1	-	1	ltp_data_present	aac_scalable_main_header()
1	-	0	max_sfb	aac_scalable_main_header()
-	-	0	ms_mask_present	aac_scalable_main_header()
1	-	0	scale_factor_grouping	aac_scalable_main_header()
-	-	0	tns_channel_mono_layer	aac_scalable_main_header()
1	-	1	tns_data_present	aac_scalable_main_header()
1	-	0	window_sequence	aac_scalable_main_header()
1	-	0	window_shape	aac_scalable_main_header()
-	0	0	common_window	channel_pair_element()
-	0	0	element_instance_tag	channel_pair_element()
-	0	0	ms_mask_present	channel_pair_element()
-	0	0	ms_used	channel_pair_element()
1	1	1	diff_control	diff_control_data()
1	1	1	diff_control_lr	diff_control_data_lr()
1	1	0	ics_reserved_bit	ics_info()
1	1	1	ltp_data_present	ics_info()
1	1	0	max_sfb	ics_info()
1	1	0	predictor_data_present	ics_info()
1	1	0	scale_factor_grouping	ics_info()
1	1	0	window_sequence	ics_info()
1	1	0	window_shape	ics_info()
1	1	1	gain_control_data_present	individual_channel_stream()
1	1	1	global_gain	individual_channel_stream()
1	1	1	length_of_longest_codeword	individual_channel_stream()
1	1	1	length_of_reordered_spectral_data	individual_channel_stream()
1	1	1	pulse_data_present	individual_channel_stream()
1	1	1	tns_data_present	individual_channel_stream()
1	-	-	element_instance_tag	lfe_channel_element()
1	1	1	ltp_coef	ltp_data()
1	1	1	ltp_lag	ltp_data()
1	1	1	ltp_lag_update	ltp_data()
1	1	1	ltp_long_used	ltp_data()
-	-	0	ms_used	ms_data()
1	1	1	number_pulse	pulse_data()
1	1	1	pulse_amp	pulse_data()
1	1	1	pulse_offset	pulse_data()
1	1	1	pulse_start_sfb	pulse_data()
4	4	4	reordered_spectral_data	reordered_spectral_data()
1	1	1	dpcm_noise_last_position	scale_factor_data()
1	1	1	dpcm_noise_nrg	scale_factor_data()
1	1	1	hcod_sf	scale_factor_data()
1	1	1	length_of_rvlc_escapes	scale_factor_data()
1	1	1	length_of_rvlc_sf	scale_factor_data()

SCE / LFE / mono layer	CPE, common_window == 0	CPE, common_window == 1 / stereo layer	data_element	function
1	1	1	rev_global_gain	scale_factor_data()
2	2	2	rvlc_cod_sf	scale_factor_data()
2	2	2	rvlc_esc_sf	scale_factor_data()
1	1	1	sf_concealment	scale_factor_data()
1	1	1	sf_escapes_present	scale_factor_data()
1	1	1	sect_cb	section_data()
1	1	1	sect_len_incr	section_data()
1	-	-	element_instance_tag	single_channel_element()
4	4	4	hcod	spectral_data()
4	4	4	hcod_esc_y	spectral_data()
4	4	4	hcod_esc_z	spectral_data()
4	4	4	pair_sign_bits	spectral_data()
4	4	4	quad_sign_bits	spectral_data()
3	3	3	coef	tns_data()
3	3	3	coef_compress	tns_data()
3	3	3	coef_res	tns_data()
3	3	3	direction	tns_data()
3	3	3	length	tns_data()
3	3	3	n_filt	tns_data()
3	3	3	order	tns_data()

Table 4.130 – AAC error sensitivity category assignment for extended payload

extension_payload	data_element	function
6	drc_band_top	dynamic_range_info()
6	drc_bands_incr	dynamic_range_info()
6	drc_bands_present	dynamic_range_info()
6	drc_bands_reserved_bits	dynamic_range_info()
6	drc_tag_reserved_bits	dynamic_range_info()
6	dyn_rng_ct	dynamic_range_info()
6	dyn_rng_sgn	dynamic_range_info()
6	excluded_chns_present	dynamic_range_info()
6	pce_instance_tag	dynamic_range_info()
6	pce_tag_present	dynamic_range_info()
6	prog_ref_level	dynamic_range_info()

6	prog_ref_level_present	dynamic_range_info()
6	prog_ref_level_reserved_bits	dynamic_range_info()
6	additional_excluded_chns	excluded_channels()
6	exclude_mask	excluded_channels()
5	extension_type	extension_payload()
5	data_element_version	extension_payload()
7	fill_byte	extension_payload()
7	fill_nibble	extension_payload()
7	other_bits	extension_payload()
8	dataElementLengthPart	extension_payload()
8	data_element_byte	extension_payload()
9	bs_sbr_crc_bits	sbr_extension_data()
9	bs_header_flag	sbr_extension_data()
9	bs_fill_bits	sbr_extension_data()
9	bs_amp_res	sbr_header()
9	bs_start_freq	sbr_header()
9	bs_stop_freq	sbr_header()
9	bs_xover_band	sbr_header()
9	bs_reserved	sbr_header()
9	bs_header_extra_1	sbr_header()
9	bs_header_extra_2	sbr_header()
9	bs_freq_scale	sbr_header()
9	bs_alter_scale	sbr_header()
9	bs_noise_bands	sbr_header()
9	bs_limiter_bands	sbr_header()
9	bs_limiter_gains	sbr_header()
9	bs_interpol_freq	sbr_header()
9	bs_smoothing_mode	sbr_header()
9	bs_data_extra	sbr_single_channel_element()
9	bs_reserved	sbr_single_channel_element()
9	bs_add_harmonic_flag	sbr_single_channel_element()
9	bs_extended_data	sbr_single_channel_element()
9	bs_extension_size	sbr_single_channel_element()
9	bs_esc_count	sbr_single_channel_element()
9	bs_extension_id	sbr_single_channel_element()
9	bs_data_extra	sbr_channel_pair_element()
9	bs_reserved	sbr_channel_pair_element()
9	bs_coupling	sbr_channel_pair_element()
9	bs_add_harmonic_flag	sbr_channel_pair_element()
9	bs_extended_data	sbr_channel_pair_element()
9	bs_extension_size	sbr_channel_pair_element()
9	bs_esc_count	sbr_channel_pair_element()
9	bs_extension_id	sbr_channel_pair_element()
9	bs_frame_class	sbr_grid()
9	tmp	sbr_grid()
9	bs_freq_res	sbr_grid()
9	bs_pointer	sbr_grid()
9	bs_var_bord_0	sbr_grid()
9	bs_var_bord_1	sbr_grid()
9	bs_num_rel_0	sbr_grid()
9	bs_num_rel_1	sbr_grid()
9	bs_df_env	sbr_dtdf()
9	bs_df_noise	sbr_dtdf()
9	bs_invf_mode	sbr_invf()
9	bs_env_start_value_balance	sbr_envelope()
9	bs_env_start_value_level	sbr_envelope()
9	bs_codeword	sbr_envelope()

9	bs_noise_start_value_balance	sbr_noise()
9	bs_noise_start_value_level	sbr_noise()
9	bs_codeword	sbr_noise()
9	bs_add_harmonic	sbr_sinusoidal_coding()

4.5.5 Figures

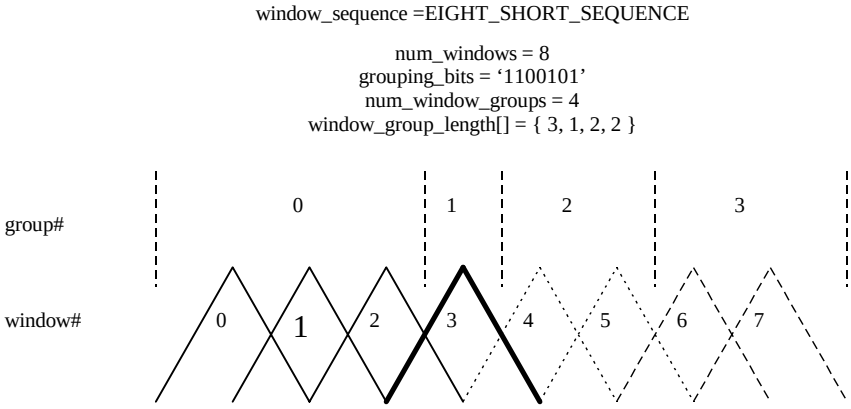


Figure 4.22 – Example for short window grouping

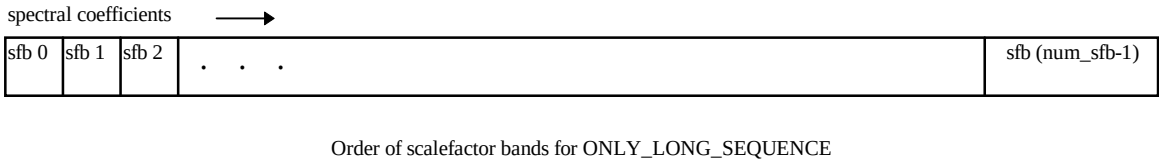


Figure 4.23 – Spectral order of scalefactor bands in case of ONLY_LONG_SEQUENCE

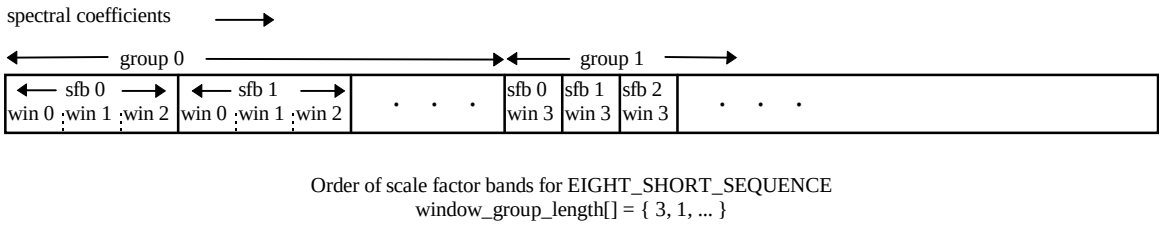


Figure 4.24 – Spectral order of scalefactor bands in case of EIGHT_SHORT_SEQUENCE

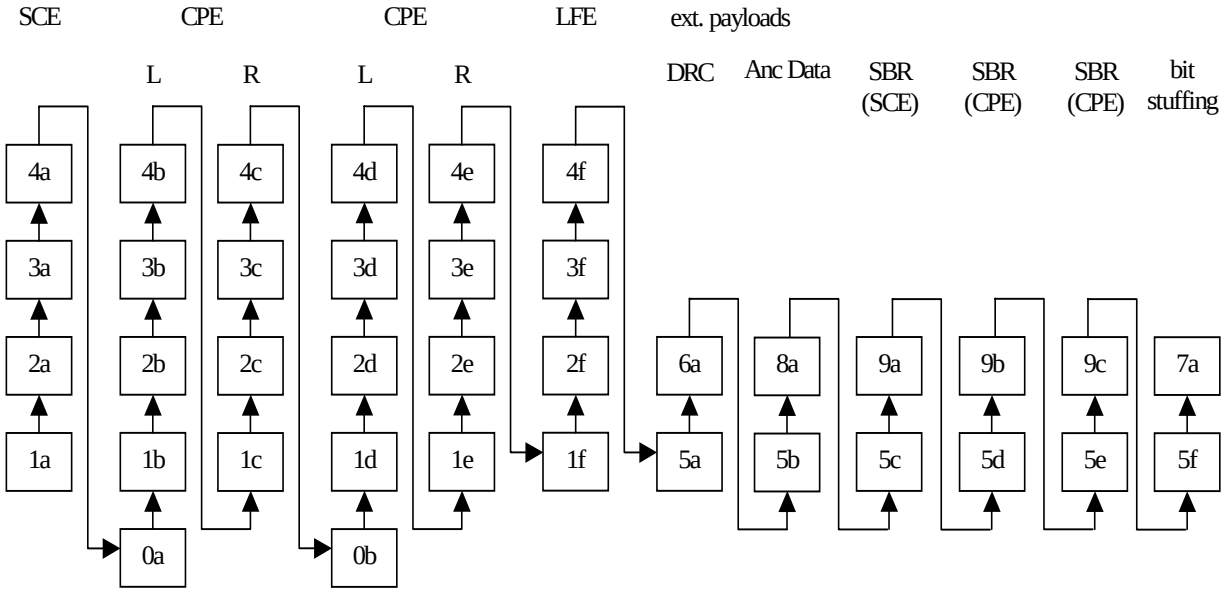


Figure 4.25 – Dependency structure in case of error resilient multichannel AAC syntax (channelConfiguration == 6)

4.6 GA-Tool Descriptions

4.6.1 Quantization

(subclause identical to ISO/IEC 13818-7)

4.6.1.1 Tool description

For quantization of the spectral coefficients in the encoder a non uniform quantizer is used. Therefore the decoder must perform the inverse non uniform quantization after the Huffman decoding of the scalefactors (see subclauses 4.6.3 and 4.6.2) and spectral data (see subclause 4.6.3).

4.6.1.2 Definitions

Help elements:

x_quant[g][win][sfb][bin]	quantized spectral coefficient for group *g*, window *win*, scalefactor band *sfb*, coefficient *bin*.
x_invquant[g][win][sfb][bin]	spectral coefficient for group *g*, window *win*, scalefactor band *sfb*, coefficient *bin* after inverse quantization.

4.6.1.3 Decoding process

The inverse quantization is described by the following formula:

$$x\_invquant = Sign(x\_quant) \cdot \left| x\_quant \right|^{\frac{4}{3}}$$

The maximum allowed absolute amplitude for x_quant is 8191. The inverse quantization is applied as follows:

```
for (g = 0; g < num_window_groups; g++) {
   for (sfb = 0; sfb < max_sfb; sfb++) {
      width = (swb_offset [sfb+1] - swb_offset [sfb]);
      for (win = 0; win < window_group_len[g]; win++) {
         for (bin = 0; bin < width; bin++) {
            x_invquant[g][win][sfb][bin] = sign(x_quant[g][win][sfb][bin]) *
                                                abs(x_quant[g][win][sfb][bin]) ^(4/3);
         }
      }
   }
```

```
}
```

4.6.2 Scalefactors

(subclause similar to ISO/IEC 13818-7)

4.6.2.1 Tool description

The basic method to adjust the quantization noise in the frequency domain is the noise shaping using scalefactors. For this purpose the spectrum is divided in several groups of spectral coefficients called scalefactor bands which share one scalefactor (see subclause 4.5.2.3.4). A scalefactor represents a gain value which is used to change the amplitude of all spectral coefficients in that scalefactor band. This mechanism is used to change the allocation of the quantization noise in the spectral domain generated by the non uniform quantizer.

For window_sequences which contain SHORT_WINDOWs grouping can be applied, i.e. a specified number of consecutive SHORT_WINDOWs may have only one set of scalefactors. Each scalefactor is then applied to a group of scalefactor bands corresponding in frequency (see subclause 4.5.2.3.4).

In this tool the scalefactors are applied to the inverse quantized coefficients to reconstruct the spectral values.

4.6.2.2 Definitions

4.6.2.2.1 Data elements

global_gain	An 8-bit unsigned integer value representing the value of the first scalefactor. It is also the start value for the following differential coded scalefactors
scale_factor_data()	Part of bitstream payload which contains the differential coded scalefactors
hcod_sf[]	Huffman codeword from the Huffman code table used for coding of scalefactors, see subclause 4.6.3.2

4.6.2.2.2 Help elements

dpcm_sf[g][sfb]	Differential coded scalefactor of group g, scalefactor band sfb.
x_rescal[]	rescaled spectral coefficients
sf[g][sfb]	Array for scalefactors of each group
get_scale_factor_gain()	Function that returns the gain value corresponding to a scalefactor

4.6.2.3 Decoding process

4.6.2.3.1 Scalefactor bands

Scalefactors are used to shape the quantization noise in the spectral domain. For this purpose, the spectrum is divided into several scalefactor bands (see subclause 4.5.2.3.4). Each scalefactor band has a scalefactor, which represents a certain gain value which has to be applied to all spectral coefficients in this scalefactor band. In case of EIGHT_SHORT_SEQUENCE a scalefactor band may contain multiple scalefactor window bands of consecutive SHORT_WINDOWs (see subclause 4.5.2.3.4 and 4.5.2.3.5).

4.6.2.3.2 Decoding of scalefactors

For all scalefactors the difference to the preceeding value is coded using the Huffman code book given in Table 4.A.1.See subclause 4.6.3 for a detailed description of the Huffman decoding process. The start value is given explicitly as a 8 bit PCM in the data element **global_gain**. A scalefactor is not transmitted for scalefactor bands which are coded with the Huffman codebook ZERO_HCB. If the Huffman codebook for a scalefactor band is coded with INTENSITY_HCB or INTENSITY_HCB2, the scalefactor is used for intensity stereo (see subclauses 4.6.3 and 4.6.8.1.4). In that case a normal scalefactor does not exist (but is initialized to zero to have an valid entry in the array).

The following pseudo code describes how to decode the scalefactors sf[g][sfb]:

```
last_sf = global_gain;
for (g = 0; g < num_window_groups; g++) {
```

```
    for (sfb = 0; sfb < max_sfb; sfb++) {
        if (sfb_cb[g][sfb] != ZERO_HCB && sfb_cb[g][sfb] != INTENSITY_HCB
            && sfb_cb[g][sfb] != INTENSITY_HCB2 ) {
            dpcm_sf = decode_huffman() - index_offset;   /* see subclause 4.6.3
                                                            „Noiseless Coding“ */
            sf[g][sfb] = dpcm_sf + last_sf;
            last_sf = sf[g][sfb];
        }
        else {
            sf[g][sfb] = 0;
        }
    }
}
```

Note that scalefactors, sf[g][sfb], must be within the range of zero to 255, both inclusive.

In case of error resilient scalefactor coding, a RVLC has been used instead of a Huffman code. The decoding process of the RVLC words is the same as for the Huffman codewords; just another codebook has to be used. This codebook uses symmetric codewords. Due to this it is possible to detect errors, because asymmetric codewords are illegal. Furthermore, decoding can be started at both sides. To allow backward decoding, an additional value is available within the bitstream payload, which contains the last scalefactor value. In case of intensity an additional codeword is available, which allows backwards decoding. In case of PNS an additional DPCM value is available for the same reason.

In the case of sf_escapes_present==1, a decoded value of ±7 is used as ESC_FLAG. It signals that an escape value exists, that has to be added to +7 or subtracted from -7 in order to find the actual scalefactor value. This escape value is Huffman encoded.

4.6.2.3.3 Applying scalefactors

The spectral coefficients of all scalefactor bands which correspond to a scalefactor have to be rescaled according to their scalefactor. In case of a window sequence that contains groups of short windows all coefficients in grouped scalefactor window bands have to be scaled using the same scalefactor.

In case of window_sequences with only one window, the scalefactor bands and their corresponding coefficients are in spectral ascending order. In case of EIGHT_SHORT_SEQUENCE and grouping the spectral coefficients of grouped short windows are interleaved by scalefactor window bands. See subclause 4.5.2.3.5 for more detailed information.

The rescaling operation is done according to the following pseudo code:

```
for (g = 0; g < num_window_groups; g++) {
    for (sfb = 0; sfb < max_sfb; sfb++) {
        width = (swb_offset [sfb+1] - swb_offset [sfb] );
        for (win = 0; win < window_group_len[g]; win++) {
            gain = get_scale_factor_gain( sf[g][sfb] );
            for (k = 0; k < width; k++) {
                x_rescal[g][window][sfb][k] =
                    x_invquant[g][window][sfb][k] * gain;
            }
        }
    }
}
```

The function get_scale_factor_gain(sf[g][sfb]) returns the gain factor that corresponds to a scalefactor. The return value follows the equation:

$$gain = 2^{0.25 \cdot (sf[g][sfb] - SF\_OFFSET)}$$

The constant SF_OFFSET must be set to 100.

The following pseudo code describes this operation:

```
get_scale_factor_gain( sf[g][sfb] )  {
    SF_OFFSET = 100;
    gain = 2^(0.25 * ( sf[g][sfb] - SF_OFFSET));
    return( gain );
```

```
}
```

4.6.3 Noiseless coding

(subclause similar to ISO/IEC 13818-7)

4.6.3.1 Tool description

Noiseless coding is used to further reduce the redundancy of the scalefactors and the quantized spectrum of each audio channel.

The global_gain is coded as an 8 bit unsigned integer. The first scalefactor associated with the quantized spectrum is differentially coded relative to the global_gain value and then Huffman coded using the scalefactor codebook. The remaining scalefactors are differentially coded relative to the previous scalefactor and then Huffman coded using the scalefactor codebook.

Noiseless coding of the quantized spectrum relies on two divisions of the spectral coefficients. The first is a division into scalefactor bands that contain a multiple of 4 quantized spectral coefficients (see subclauses 4.5.2.3.4 and 4.5.2.3.5).

The second division, which is dependent on the quantized spectral data, is a division by scalefactor bands to form sections. The significance of a section is that the quantized spectrum within the section is represented using a single Huffman codebook chosen from a set of 11 possible codebooks. The length of a section and its associated Huffman codebook must be transmitted as side information in addition to the section's Huffman coded spectrum. Note that the length of a section is given in scalefactor bands rather than scalefactor window bands (see subclauses 4.5.2.3.4). In order to maximize the match of the statistics of the quantized spectrum to that of the Huffman codebooks the number of sections is permitted to be as large as the number of scalefactor bands. The maximum size of a section is max_sfb scalefactor bands.

As indicated in Table 4.132, spectrum Huffman codebooks can represent signed or unsigned n-tuples of coefficients. For unsigned codebooks, sign bits for every non-zero coefficient in the n-tuple immediately follow the associated codeword.

The noiseless coding has two ways to represent large quantized spectra. One way is to send the escape flag from the escape (ESC) Huffman codebook, which signals that the bits immediately following that codeword plus optional sign bits are an escape sequence that encodes values larger than those represented by the ESC Huffman codebook. A second way is the pulse escape method, in which relatively large-amplitude coefficients can be replaced by coefficients with smaller amplitudes in order to enable the use of Huffman code tables with higher coding efficiency. This replacement is corrected by sending the position of the spectral coefficient and the differences in amplitude as side information. The frequency information is represented by the combination of the scalefactor band number to indicate a base frequency and an offset into that scalefactor band.

4.6.3.2 Definitions

sect_cb[g][i]	spectrum Huffman codebook used for section i in group g.
sect_len_incr	used to compute the length of a section, measures number of scalefactor bands from start of section. The length of **sect_len_incr** is 3 bits if window_sequence is EIGHT_SHORT_SEQUENCE and 5 bits otherwise.
global_gain	global gain of the quantized spectrum, sent as unsigned integer value.
hcod_sf[]	Huffman codeword from the Huffman code table used for coding of scalefactors.
hcod[sect_cb[g][i]][w][x][y][z]	Huffman codeword from codebook **sect_cb[g][i]** that encodes the next 4-tuple (w, x, y, z) of spectral coefficients, where w, x, y, z are quantized spectral coefficients. Within an n-tuple, w, x, y, z are ordered as described in subclause 4.5.2.3.5. so that x_quant[group][win][sfb][bin] = w, x_quant[group][win][sfb][bin+1] = x, x_quant[group][win][sfb][bin+2] = y and x_quant[group][win][sfb][bin+3] = z. N-tuples progress from low to high frequency within the current section.
hcod[sect_cb[g][i]][y][z]	Huffman codeword from codebook **sect_cb[g][i]** that encodes the next 2-tuple (y, z) of spectral coefficients, where y, z are quantized spectral coefficients. Within an n-tuple, y, z are ordered as described in

	subclause 4.5.2.3.5 so that x_quant[group][win][sfb][bin] = y and x_quant[group][win][sfb][bin+1] = z. N-tuples progress from low to high frequency within the current section.
quad_sign_bits	sign bits for non-zero coefficients in the spectral 4-tuple. A '1' indicates a negative coefficient, a '0' a positive one. Bits associated with lower frequency coefficients are sent first.
pair_sign_bits	sign bits for non-zero coefficients in the spectral 2-tuple. A '1' indicates a negative coefficient, a '0' a positive one. Bits associated with lower frequency coefficients are sent first.
hcod_esc_y	escape sequence for quantized spectral coefficient y of 2-tuple (y,z) associated with the preceeding Huffman codeword.
hcod_esc_z	escape sequence for quantized spectral coefficient z of 2-tuple (y,z) associated with the preceeding Huffman codeword.
pulse_data_present	1 bit indicating whether the pulse escape is used (1) or not (0). Note that pulse_data_present must be 0, if window_sequence == EIGHT_SHORT_SEQUENCE;
number_pulse	2 bits indicating how many pulse escapes are used. The number of pulse escapes is from 1 to 4.
pulse_start_sfb	6 bits indicating the index of the lowest scalefactor band where the pulse escape is achieved.
pulse_offset[i]	5 bits indicating the offset.
pulse_amp[i]	4 bits indicating the unsigned magnitude of the pulse.
sect_start[g][i]	offset to first scalefactor band in section i of group g.
sect_end[g][i]	offset to one higher than last scalefactor band in section i of group g.
num_sec[g]	number of sections in group g.
escape_flag	the value of 16 in the ESC Huffman codebook
escape_prefix	the bit sequence of N 1's
escape_separator	one 0 bit
escape_word	an N+4 bit unsigned integer word, msb first
escape_sequence	the sequence of escape_prefix, escape_separator and escape_word
escape_code	2^(N+4) + escape_word
x_quant[g][win][sfb][bin]	*Huffman decoded value for group* g*, window* win*, scalefactor band* sfb*, coefficient* bin
spec[w][k]	*de-interleaved spectrum.* w *ranges from 0 to* num_windows*-1 and* k *ranges from 0 to* swb_offset[num_swb]*-1.*

The noiseless coding tool requires these constants (see Table 4.50).

ZERO_HCB	0
FIRST_PAIR_HCB	5
ESC_HCB	11
QUAD_LEN	4
PAIR_LEN	2
NOISE_HCB	13
INTENSITY_HCB2	14

INTENSITY_HCB	15
ESC_FLAG	16

4.6.3.3 Decoding process

Four-tuples or 2-tuples of quantized spectral coefficients are Huffman coded and transmitted starting from the lowest-frequency coefficient and progressing to the highest-frequency coefficient. For the case of multiple windows per block (*EIGHT_SHORT_SEQUENCE),* the grouped and interleaved set of spectral coefficients is treated as a single set of coefficients that progress from low to high. The set of coefficients may need to be de-interleaved after they are decoded (see subclause 4.5.2.3.5). Coefficients are stored in the array x_quant[g][win][sfb][bin], and the order of transmission of the Huffman codewords is such that when they are decoded in the order received and stored in the array, *bin* is the most rapidly incrementing index and *g* is the most slowly incrementing index. Within a codeword, for those associated with spectral four-tuples, the order of decoding is w, x, y, z; for codewords associated with spectral two-tuples, the order of decoding is y, z. The set of coefficients is divided into sections and the sectioning information is transmitted starting from the lowest frequency section and progressing to the highest frequency section. The spectral information for sections that are coded with the „zero“ codebook is not sent as this spectral information is zero. Similarly, spectral information for sections coded with the „intensity“ codebooks is not sent. The spectral information for all scalefactor bands at and above **max_sfb**, for which there is no section data, is zero.

There is a single differential scalefactor codebook which represents a range of values as shown in Table 4.131. The differential scalefactor codebook is shown in Table 4.A.1. There are eleven Huffman codebooks for the spectral data, as shown in Table 4.132. The codebooks are shown in Table 4.A.2 through Table 4.A.12. There are four other „codebooks“ above and beyond the actual Huffman codebooks, specifically the „zero“ codebook, indicating that neither scalefactors nor quantized data will be transmitted, and the „intensity“ codebooks indicating that this individual channel is part of a channel pair, and that the data that would normally be scalefactors is instead steering data for intensity stereo. Similarly, the „noise substitution“ codebook indicates that the spectral coefficients are derived from random numbers rather than quantized spectral values, and that the data that would normally be scalefactors is instead noise energy data. In these cases, no quantized spectral data are transmitted. Codebook index 12 is reserved.

The spectrum Huffman codebooks encode 2- or 4-tuples of signed or unsigned quantized spectral coefficients, as shown in Table 4.132. This table also indicates the largest absolute value (LAV) able to be encoded by each codebook and defines a boolean helper variable array, unsigned_cb[], that is 1 if the codebook is unsigned and 0 if signed.

The result of Huffman decoding each differential scalefactor codeword is the codeword index, listed in the first column of Table 4.A.1. This is translated to the desired differential scalefactor by adding index_offset to the index. Index_offset has a value of –60, as shown in Table 4.131. Likewise, the result of Huffman decoding each spectrum n-tuple is the codeword index, listed in the first column of Table 4.A.2 through Table 4.A.12. This index is translated to the n-tuple spectral values as specified in the following pseudo C-code:

unsigned = Boolean value unsigned_cb[i], listed in second column of Table 4.132.
dim = Dimension of codebook, listed in the third column of Table 4.132.
lav = LAV, listed in the fourth column of Table 4.132.
idx = codeword index

```
if (unsigned) {
    mod = lav + 1;
    off = 0;
}
else {
    mod = 2*lav + 1;
    off = lav;
}

if (dim == 4) {
    w = INT(idx/(mod*mod*mod)) - off;
    idx -= (w+off)*(mod*mod*mod);
    x = INT(idx/(mod*mod)) - off;
    idx -= (x+off)*(mod*mod);
    y = INT(idx/mod) - off;
    idx -= (y+off)*mod;
    z = idx - off;
```

```
}
else {
    y = INT(idx/mod) - off;
    idx -= (y+off)*mod;
    z = idx - off;
}
```

If the Huffman codebook represents signed values, the decoding of the quantized spectral n-tuple is complete after Huffman decoding and translation of codeword index to quantized spectral coefficients. If the codebook represents unsigned values then the sign bits associated with non-zero coefficients immediately follow the Huffman codeword, with a '1' indicating a negative coefficient and a '0' indicating a positive one. For example, if a Huffman codeword from codebook 7

hcod[7][y][z]

has been parsed, then immediately following this in the bitstream payload is

pair_sign_bits

which is a variable length field of 0 to 2 bits. It can be parsed directly from the bitstream payload as

```
    if (y != 0)
        if (one_sign_bit == 1)
            y = -y ;
    if (z != 0)
        if (one_sign_bit == 1)
            z = -z;
```

where one_sign_bit is the next bit in the bitstream payload and **pair_sign_bits** is the concatenation of the one_sign_bit fields.

The ESC codebook is a special case. It represents values from 0 to 16 inclusive, but values from 0 to 15 encode actual data values, and the value16 is an escape_flag that signals the presence of **hcod_esc_y** or **hcod_esc_z**, either of which will be denoted as an escape_sequence. This escape_sequence permits quantized spectral elements of LAV>15 to be encoded. It consists of an escape_prefix of N 1's, followed by an escape_separator of one zero, followed by an escape_word of N+4 bits representing an unsigned integer value. The escape_sequence has a decoded value of 2^(N+4)+escape_word. The desired quantized spectral coefficient is then the sign indicated by the pair_sign_bits applied to the value of the escape_sequence. In other words, an escape_sequence of 00000 would decode as 16, an escape_sequence of 01111 as 31, an escape_sequence of 1000000 as 32, one of 1011111 as 63, and so on. Note that restrictions in subclause 4.6.1.3 dictate that the length of the escape_sequence is always less than 22 bits. For escape Huffman codewords the ordering of data elements is Huffman codeword followed by 0 to 2 sign bits followed by 0 to 2 escape sequences.

When **pulse_data_present** is 1 (the pulse escape is used), one or several quantized coefficients have been replaced by coefficients with smaller amplitudes in the encoder. The number of coefficients replaced is indicated by **number_pulse**. In reconstructing the quantized spectral coefficients x_quant this replacement is compensated by adding **pulse_amp** to or subtracting **pulse_amp** from the previously decoded coefficients whose frequency indices are indicated by **pulse_start_sfb** and **pulse_offset**. Note that the pulse escape method is illegal for a block whose **window_sequence** is *EIGHT_SHORT_SEQUENCE*. The decoding process is specified in the following pseudo-C code:

```
if (pulse_data_present) {
    g = 0;
    win = 0;
    k = swb_offset[pulse_start_sfb];
    for (j = 0; j < number_pulse+1; j++) {
        k += pulse_offset[j];

        /* translate_pulse_parameters(); */
        for (sfb = pulse_start_sfb; sfb<num_swb;sfb++) {
            if (k < swb_offset[sfb+1]) {
                bin = k - swb_offset[sfb] ;
                break;
            }
        }

        /* restore coefficients */
        if (x_quant[g][win][sfb][bin] > 0 )
            x_quant[g][win][sfb][bin] += pulse_amp[j];
```

```
        else
            x_quant[g][win][sfb][bin] -= pulse_amp[j];
    }
}
```

Several decoder tools (TNS, filterbank) access the spectral coefficients in a non-interleaved fashion, i.e. all spectral coefficients are ordered according to window number and frequency within a window. This is indicated by using the notation spec[w][k] rather than x_quant[g][w][sfb][bin].

The following pseudo C-code indicates the correspondence between the four-dimensional, or interleaved, structure of array x_quant[ ][ ][ ][ ] and the two-dimensional, or de-interleaved, structure of array spec[ ][ ]. In the latter array the first index increments over the individual windows in the window sequence, and the second index increments over the spectral coefficients that correspond to each window, where the coefficients progress linearly from low to high frequency.

```
quant_to_spec() {
    k = 0;
    for (g = 0; g < num_window_groups; g++ ) {
        j = 0;
        for (sfb = 0; sfb < num_swb; sfb ++) {
            width = swb_offset[sfb+1] - swb_offset[sfb];
            for (win = 0; win < window_group_length[g]; win++) {
                for (bin=0; bin < width; bin++) {
                    spec[win+k][bin+j] = x_quant[g][win][sfb][bin] ;
                }
            }
            j+=width;
        }
        k+=window_group_length[g];
    }
}
```

Decoding of reordered spectral data cannot be done straightforward. The following c-like description shows the decoding process:

```
/* helper functions */
void InitReordering(void);
  /* Initializes variables used by the reordering functions like the segment
     widths and the used offsets in segments and codewords */

void InitRemainingBitsInSegment(void);
  /* Initializes remainingBitsInSegment[] array for each segment with the
     total size of the segment */

int  DecodeCodeword(codewordNr, segmentNr, direction);
  /* Try to decode the codeword indexed by codewordNr using data already read
     for this codeword and using data from the segment index by segmentNr.
     The read direction in the segment is given by direction.
     DecodeCodeword returns the number of bits read from the indexed segment. */

void MoveFromSegmentToCodeword(codewordNr, segmentNr, bitLen, direction);
  /* Move bitLen bits from the segment indexed by segmentNr to the codeword
     indexed by codewordNr using direction as read direction in the segment.
     The bits are appended to existing bits for the codeword and the codeword
     length is adjusted. */

void AdjustOffsetsInSegment(segmentNr, bitLen, direction);
  /* Like MoveFromSegmentToCodeword(), but no bits are moved. Only the offsets
     for the segment indexed by segmentNr are adjusted according bitLen and
     direction. */

void MarkCodewordAsDecoded(codewordNr);
  /* Marks the codeword indexed by codewordNr as decoded. */

bool CodewordIsNotDecoded(codewordNr);
  /* Returns TRUE if the codeword indexed by codewordNr is not decoded. */

void ToggleReadDirection(void);
  /* Toggles the read direction in the segments between forward and backward. */
```

```
/* (input) variables */
numberOfCodewords;
numberOfSegments;
numberOfSets;

DecodeReorderedSpectralData()
{
  InitReordering();
  InitRemainingBitsInSegment();

  /* first step: decode PCWs (set 0) */
  readDirection = forward;
  for (codeword = 0; codeword < numberOfSegments; codeword++) {
    cwLen = DecodeCodeword(codeword, codeword, readDirection);
    if (cwLen <= remainingBitsInSegment[codeword]) {
      AdjustOffsetsInSegment(codeword, cwLen, readDirection);
      MarkCodewordAsDecoded(codeword);
      remainingBitsInSegment[codeword] -= cwLen;
    }
    else {
      /* error !!! (PCWs do always fit into segments) */
    }
  }

  /* second step: decode nonPCWs */
  for (set = 1; set < numberOfSets; set++) {
    ToggleReadDirection();
    for (trial = 0; trial < numberOfSegments; trial++) {
      for (codewordBase = 0; codewordBase < numberOfSegments; codewordBase++) {
        segment  = (trial + codewordBase) % numberOfSegments;
        codeword = codewordBase + set*numberOfSegments;

        if (CodewordIsNotDecoded(codeword) &&
            (remainingBitsInSegment[segment] > 0)) {
          cwLenInSegment = DecodeCodeword(codeword, segment, readDirection);
          if (cwLenInSegment <= remainingBitsInSegment[segment]) {
            AdjustOffsetsInSegment(segment, cwLenInSegment, readDirection);
            MarkCodewordAsDecoded(codeword);
            remainingBitsInSegment[segment] -= cwLenInSegment;
          }
          else {  /* only part of codeword in segment */
            MoveFromSegmentToCodeword(codeword,
                                      segment,
                                      remainingBitsInSegment[segment],
                                      readDirection);
            remainingBitsInSegment[segment] = 0;
          }
        }
      }
    }
  }
}
```

4.6.3.4 Tables

Table 4.131 – Scalefactor Huffman codebook parameters

Codebook Number	Dimension of Codebook	index_offset	Range of values	Codebook listed in Table
0	1	-60	-60 to +60	Table 4.A.1

Table 4.132 – Spectrum Huffman codebooks parameters

Codebook number, i	unsigned_cb[i]	Dimension of codebook	LAV for codebook	Codebook listed in Table
0	-	-	0	-

1	0	4	1	Table 4.A.2
2	0	4	1	Table 4.A.3
3	1	4	2	Table 4.A.4
4	1	4	2	Table 4.A.5
5	0	2	4	Table 4.A.6
6	0	2	4	Table 4.A.7
7	1	2	7	Table 4.A.8
8	1	2	7	Table 4.A.9
9	1	2	12	Table 4.A.10
10	1	2	12	Table 4.A.11
11	1	2	16 (with ESC 8191)	Table 4.A.12
12	-	-	(reserved)	-
13	-	-	perceptual noise substitution	-
14	-	-	intensity out-of-phase	-
15	-	-	intensity in-phase	-
16	1	2	16 (w/o ESC 15)	Table 4.A.12
17	1	2	16 (with ESC 31)	Table 4.A.12
18	1	2	16 (with ESC 47)	Table 4.A.12
19	1	2	16 (with ESC 63)	Table 4.A.12
20	1	2	16 (with ESC 95)	Table 4.A.12
21	1	2	16 (with ESC 127)	Table 4.A.12
22	1	2	16 (with ESC 159)	Table 4.A.12
23	1	2	16 (with ESC 191)	Table 4.A.12
24	1	2	16 (with ESC 223)	Table 4.A.12
25	1	2	16 (with ESC 255)	Table 4.A.12
26	1	2	16 (with ESC 319)	Table 4.A.12
27	1	2	16 (with ESC 383)	Table 4.A.12
28	1	2	16 (with ESC 511)	Table 4.A.12
29	1	2	16 (with ESC 767)	Table 4.A.12
30	1	2	16 (with ESC 1023)	Table 4.A.12
31	1	2	16 (with ESC 2047)	Table 4.A.12

4.6.4 Noiseless coding for the fine grain scalability

4.6.4.1 Tool description

BSAC stands for bit sliced arithmetic coding and is the name of a noiseless coder and bitstream payload formatter that provides a fine grain scalability and error resilience in the MPEG-4 General Audio (GA) coder. The BSAC noiseless coding module is an alternative to the AAC coding module, with all other modules of the AAC-based coder remaining unchanged. The BSAC noiseless coding is used to make the bitstream payload scalable and error resilience and further reduce the redundancy of the scalefactors and the quantized spectrum. The BSAC noiseless decoding process is split into 4 subclauses. Subclause 4.6.4.2 to 4.6.4.6 describes the detailed decoding process of the spectral data, the stereo or pns related data, the scalefactors and the coding band side information.

4.6.4.2 Decoding of bit sliced spectral data (bsac_spectral_data)

4.6.4.2.1 Description

BSAC uses the bit-slicing scheme of the quantized spectral coefficients in order to provide the fine grain scalability. And it encode the bit-sliced data using binary arithmetic coding scheme in order to reduce the average bits transmitted while suffering no loss of fidelity.

In BSAC scalable coding scheme, a quantized sequence is divided into coding bands, as shown in subclause 4.5.2.6.2.5. And, a quantized sequence is mapped into a bit-sliced sequence within a coding band. The noiseless coding of the sliced bits relies on the probability table of the coding band, the significance and the other contexts.

The significance of the bit-sliced data is the position of the sliced bit to be coded.

The flags, sign_is_coded[] are updated with coding the vectors from MSB to LSB. They are initialized to 0. And they are set to 1 when the sign of the quantized spectrum is coded.

The probability table for encoding the bit-sliced data within each coding band is included in the bistream element **cband_si_type** and transmitted starting from the lowest coding band and progressing to the highest coding band allocated to each layer. For the detailed description of the coding band side information **cband_si_type**, see subclause 4.6.4.5. Table 4.A.31 lists 23 probability tables which are used for encoding/decoding the bit-sliced data. The BSAC probability table consists of several sub-tables. Sub-tables are classified and chosen according to the significance and the coded upper bits as shown Table 4.A.54 to Table 4.A.75. Every sliced bit is arithmetic encoded using the probability value chosen among several possible sub-tables of BSAC probability table.

4.6.4.2.2 Definitions

4.6.4.2.2.1 Data elements

acod_sliced_bit[ch][g][i]	Arithmetic codeword necessary for arithmetic decoding of the sliced bit. Using this decoded bit, we can reconstuct each bit value of the quantized spectral value. The actually reconstructed bit-value is depentent on the significance of the sliced bit.
acod_sign[ch][g][i]	Arithmetic codeword from binary arithmetic coding sign_bit. The probability of the “0” symbol is defined to 0.5 which uses 8192 as a 14-bit fixed-point number. sign_bit indicates sign bit for non-zero coefficient. A “1” indicates a negative coefficient, a “0” a positive one. When the bit value of the quantized signal is assigned 1 for the first time, sign bit is arithmetic coded and sent.

4.6.4.2.2.2 Help elements

layer	scalability layer index
snf	significance of vector to be decoded.
ch	channel index
nch	the number of channel
cur_snf [i]	current significance of the i-th vector. cur_snf[] is initialized to Abit[cband]. See subclause 4.5.2.6.2.5.
maxsnf	maximum of current significance of the vectors to be decoded. See subclause 4.5.2.6.2.5.
snf	significance index
layer_data_available()	function that returns “1” as long as each layer’s bitstream payload is available, otherwise “0”. In other words, it indicates whether the remaing bitstream payload of each layer is available or not.
layer_group[layer]	indicates the group index of the spectral data to be added newly in the scalability layer. See subclause 4.5.2.6.2.5
layer_start_index[layer]	indicates the index of the lowest spectral component to be added newly in the scalability layer. See subclause 4.5.2.6.2.5
layer_end_index[layer]	indicates the index of the highest spectral component to be added newly in the scalability layer. See subclause 4.5.2.6.2.5
start_index[g]	indicates the index of the lowest spectral component to be coded in the group g
end_index[g]	indicates the index of the lowest spectral component to be coded in the group g
sliced_bit	the decoded value of the sliced bits of the quantized spectrum.
sample[ch][g][i]	quantized spectral coefficients reconstructed from the decoded bit-sliced data of spectral line i in channel ch and group index g. See subclause 4.5.2.6.2.2

sign_is_coded[ch][g][i]	flag that indicates whether the sign of the ith quantized spectrum is already coded (1) or not (0) in channel ch and group index g.
sign_bit[ch][g][i]	sign bit for non-zero coefficient. A “1” indicates a negative coefficient, a “0” a positive one. When the bit value of the quantized signal is assigned 1 for the first time, sign bit is arithmetic coded and sent.

4.6.4.2.3 Decoding process

In BSAC encoder, the absolute values of quantized spectral coefficients is mapped into a bit-sliced sequence. These slicded bits are the symbols of the arithmetic coding. Every siliced bits are binary arithmetic coded from the lowest-frequency coefficient to the highest-frequency coefficient of the scalability layer, starting the Most Significant Bit(MSB) plane and progressing to the Least Significant Bit(LSB) plane. The arithmetic coding of the sign bits associated with non-zero coefficient follows that of the sliced bit when the bit-slice of the spectral coefficient is 1 for the first time.

For the case of multiple windows per block, the concatenated and possibly grouped and interleaved set of spectral coefficients is treated as a single set of coefficients that progress from low to high as described in subclause 4.5.2.6.2.6. This set of spectral coefficients may need to be de-interleaved after they are decoded. The spectral information for all scalefactor bands equal to or greater than max_sfb is set to zero.

After all MSB data are encoded from the lowest frequency line to the highest, the same encoding process is repeated until LSB data is encoded or the layer data is not available.

The length of the available bitstream payload (*available_len[]*) is initialized at the beginning of each layer as described in subclause 4.5.2.6.2.5. The estimated length of the codeword (*est_cw_len*) to be decoded is calculated from the arithmetic decoding process as described in subclause 4.5.2.6.2.7. After the arithmetic decoding of a symbol, the length of the available bitstream payload should be updated by subtracting the estimated codeword length from it. We can detect whether the remaing bitstream payload of each layer is available or not by checking the available_len.

The bit-sliced data is decoded with the probability, which is selected among values listed in from Table 4.A.54 to Table 4.A.75.

The probability value should be defined in order to arithmetic-code the symbols (the sliced bits). Binary probability table is made up of probability values (p0) of the symbol ‘0’. First of all, probability table is selected using cband_si as shown in Table 4.A.29. Next, the sub-table is selected in the probability table according to the context such as the current significance of the spectral coefficient and the higher bit-slices that have been decoded. All the vector of the higher bit-slices, *higher_bit_vector* are initialized to 0 before the coding of the bit-sliced data is started. Whenever the bit-slice is coded, the vector, *higher_bit_vector* is updated as follows:

```
higher_bit_vector[ch][g][i] = (higher_bit_vector[ch][g][i]<<1) + decoded_bitslice;
if(higher_bit_vector[ch][g][i]) {
     if (higher_bit_vector[ch][g][i] > 15)
         p0_index = 15;
     else
         p0_index = higher_bit_vector[ch][g][i] -1;
}
```

And, the probability (p0) is selected among the several values in the sub-table. In order to select one of the several probability values in the sub-table, the index of the probability should be decided. If the higher bit-slice vector is non-zero, the index of the probability (p0) is (higher_bit_vector[ch][g][i] - 1). Otherwise, it relies upon the sliced bits of successive non-overlapping 4 spectral data as shown in Table 4.A.32.

However if the available codeword size is smaller than 14, there is a constraints on the selected probability value as follows:

```
   if (available_len <14) {
       if (p0 < min_p0[available_len])
           p0 = min_p0[available_len];
       else if (p0 > max_p0[available_len])
           p0 = max_p0[available_len];
   }
```

The minimum probability min_p0[] and the maximum probability max_p0[] is listed in Table 4.A.33 and Table 4.A.34.

Detailed arithmetic decoding procedure is described in this subclause 4.5.2.6.2.7.

There are 23 probability tables which can be used for encoding/decoding the bit-sliced data. 23 probability table are provided to cover the different statistics of the bit-slices. In order to transmit the probability table used in encoding process, the probability table is included in the syntax element, **cband_si**. After **cband_si** is decoded, the probaiblity table is mapped from **cband_si** using Table 4.A.31 and the decoding of the bit-sliced data shall be started.

The current significance of the spectral coefficient represents the bit-plane of the bit-slice to be decoded. Table 4.A.31 shows the MSB plane of the decoded sample according to **cband_si**. Current significance, *cur_snf[]* of all spectral coefficient within a coding band are initialized to the MSB plane. For the detailed initialization process, see subclause 4.5.2.6.2.5

The arithmetic decoding of the sign bit associated with non-zero coefficient follows the arithmetic decoding of the sliced bit when the bit-value of the quantized spectral coefficient is 1 for the first time, with a 1 indicating a negative coefficient and a 0 indicating a positive one. The flag, *sign_is_coded[]* represents whether the sign bit of the quantized spectrum has been decoded or not. Before the decoding of the bit-sliced data is started, all the *sign_is_coded* flags are set to 0. The flag, *sign_is_coded* is set to 1 after the sign bit is decoded. The decoding process of the sign bit can be summarized as follows:

```
i = the spectral line index
if (sample[ch][g][i] && !sign_is_coded[ch][g][i]) {
   arithmetic decoding of the sign bit;
   sign_is_coded[ch][g][i] = 1;
}
```

Decoded symbol need to be reconstructed to the sample. For the detailed reconstruction of the blt-sliced data, see **Reconstruction of the decoded sample from bit-sliced data** part in subclause 4.5.2.6.2.2.

4.6.4.3 Decoding of stereo_info, ms_used and noise_flag

4.6.4.3.1 Descriptions

The BSAC scalable coding scheme includes the noiseless coding which is different from MPEG-4 AAC coding and further reduce the redundancy of the stereo-related data.

Decoding of the stereo-related data and Perceptual Noise Substitution(pns) data is depended on pns_data_present and stereo_info which indicates the stereo mask. Since the decoded data is the same value with MPEG-4 AAC, the MPEG-4 AAC stereo-related and pns processing follows the decoding of the stereo-related data and pns data.

4.6.4.3.2 Definitions

4.6.4.3.2.1 Data elements

acode_ms_used[g][sfb]	arithmetic codeword from the arithmetic coding of ms_used which is one-bit flag per scalefactor band indicating that M/S coding is being used in window group g and scalefactor band sfb, as follows: 0 Independent 1 ms_used
acode_stereo_info[g][sfb]	arithmetic codeword from the arithmetic coding of stereo_info which is two-bit flag per scalefactor band indicating that M/S coding or Intensity coding is being used in window group g and scalefactor band sfb, as follows: 00 Independent 01 ms_used 10 Intensity_in_phase 11 Intensity_out_of_phase or noise_flag_is_used Note : If ms_mask_present is 3, noise_flag_l and noise_flag_r are 0 value, then stereo_info is interpreted as out-of-phase intensity stereo regardless the value of pns_data_present.
acode_noise_flag[g][sfb]	arithmetic codeword from the arithmetic coding of noise_flag which is 1-bit flag per scalefactor band indicating whether the perceptual noise substitution is used(1) or not(0) in window group g and scalefactor band sfb.

acode_noise_flag_l[g][sfb]	arithmetic codeword from the arithmetic coding of noise_flag_l which is 1-bit flag per scalefactor band indicating whether the perceptual noise substitution is used(1) or not(0) in the left channel, window group g and scalefactor band sfb .
acode_noise_flag_r[g][sfb]	arithmetic codeword from the arithmetic coding of noise_flag which is 1-bit flag per scalefactor band indicating whether the perceptual noise substitution is used(1) or not(0) in the right channel, window group g and scalefactor band sfb.
acode_noise_mode[g][sfb]	arithmetic codeword from the arithmetic coding of noise_mode which is two-bit flag per scalefactor band indicating that which noise substitution is being used in window group g and scalefactor band sfb, as follows: 00 Noise Subst L+R (independent) 01 Noise Subst L+R (correlated) 10 Noise Subst L+R (correlated, out-of-phase) 11 reserved

4.6.4.3.2.2 Help elements

ch	channel index
g	group index
sfb	scalefacotor band index within group
layer	scalability layer index
nch	the number of channel
ms_mask_present	this two bit field indeicates that the stereo mask is 00 Independent 01 1 bit mask of ms_used is located in the layer sfb side information part. 10 All ms_used are ones 11 2 bit mask of stereo_info is located in the layer sfb side information part.
layer_group[layer]	indicates the group index of the spectral data to be added newly in the scalability layer. See subclause 4.5.2.6.2.5
layer_start_sfb[layer]	indicates the index of the lowest scalefactor band index to be added newly in the scalability layer. See subclause 4.5.2.6.2.5
layer_end_sfb[layer]	indicates the highest scalefactor band index to be added newly in the scalability layer. See subclause 4.5.2.6.2.5

4.6.4.3.3 Decoding process

Decoding process of ms_mask_present, noise_flag or ms_used is depended on pns_data_present, number of channel and ms_mask_present. pns_data_present flag is conveyed as a element in syntax of general_header(). pns_data_present indicates whether pns tool is used or not at each frame. stereo_info indeicates the stereo mask as follows :

00 Independent

01 1 bit mask of ms_used is located in the layer sfb side information part.

10 All ms_used are ones

11 2 bit mask of stereo_info is located in the layer sfb side information part.

Detailed arithmetic decoding procedure is described in this subclause 4.5.2.6.2.7.

Decoding process is classified as follows :

- 1 channel, no pns data

  If the number of channel is 1 and pns data is not present, there is no data elements related to stereo or pns.

- 1 channel, pns data

  If the number of channel is 1 and pns data is present, noise flag of the scalefactor bands between **pns_start_sfb** to **max_sfb** is arithmetic decoded using model shown in Table 4.A.52. Perceptual noise substitution is done according to the decoded noise flag.

- 2 channel, ms_mask_present=0 (Independent), No pns data

  If ms_mask_present is 0 and pns data is not present, arithmetic decoding of stereo_info or ms_used is not needed.

- 2 channel, ms_mask_present=0 (Independent), pns data

  If ms_mask_present is 0 and pns data is present, noise flag for pns is arithmetic decoded using model shown in Table 4.A.52. Perceptual noise substitution of independent mode is done according to the decoded noise flag.

- 2 channel, ms_mask_present=2 (all ms_used), pns data or no pns data

  All ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band. And naturally there can be no pns processing regardless of pns_data_present flag.

- 2 channel, ms_mask_present=1 (optional ms_used), pns data or no pns data

  1 bit mask of max_sfb bands of ms_used is conveyed in this case. So, ms_used is arithmetic decoded using the ms_used model given in Table 4.A.50. M/S stereo processing of AAC is done or not according to the decoded ms_used. If ms_used is 1, there is no pns processing.

- 2 channel, ms_mask_present=3 (optional ms_used/intensity/pns), no pns data

  At first, stereo_info is arithmetic decoded using the stereo_info model given in Table 4.A.51.

  stereo_info is is two-bit flag per scalefactor band indicating that M/S coding or Intensity coding is being used in window group g and scalefactor band sfb as follows :

  00 Independent

  01 ms_used

  10 Intensity_in_phase

  11 Intensity_out_of_phase

  If stereo_info is not 0, M/S stereo or intensity stereo of AAC is done with these decoded data. Since pns data is not present, we don_t have to process pns.

- 2 channel, ms_mask_present=3 (optional ms_used/intensity/pns), pns data

  stereo_info is arithmetic decoded using the stereo_info model given in Table 4.A.51.

  If stereo_info is 1 or 2, M/S stereo or intensity stereo processing of AAC is done with these decoded data and there is no pns processing.
  If stereo_info is 3 and scalefactor band is larger than or equal to pns_start_sfb, noise flag for pns is arithmetic decoded using model given in Table 4.A.52. And then if the both noise flags of two channel are 1, noise substitution mode is arithmetic decoded using model given in Table 4.A.53. The perceptual noise is substituted or out_of_phase intensity stereo processing is done according to the substitution mode. Otherwise, the perceptual noise is substituted only if noise flag is 1.

  If stereo_info is 3 and scalefactor band is smaller than pns_start_sfb, out_of_phase intensity stereo processing is done.

4.6.4.4 Decoding of scalefactors, noise energy and intensity stereo position

4.6.4.4.1 Description

The BSAC scalable coding scheme includes the noiseless coding which is different from AAC and further reduce the redundancy of the scalefactors.

The max_scalefactor is coded as an 8 bit unsigned integer. The scalefactors are differentially coded relative to the max_scalefactor value and then Arithmetic coded using the differential scalefactor model.

4.6.4.4.2 Definitions

4.6.4.4.2.1 Data elements

acode_scf_index[ch][g][sfb]	Arithmetic codeword from the coding of the differential scalefactors.
acode_max_noise_energy [ch]	Arithmetic codeword from the coding of the maximum of the noise energies.
acode_dpcm_noise_energy_index[ch][g][sfb]	Arithmetic codeword from the coding of the differential noise energy index.
acode_is_position_index[g][sfb]	Arithmetic codeword from the coding of the intensity stereo poistion index.

4.6.4.4.2.2 Help elements

ch	channel index
g	group index
sfb	scalefacotor band index within group
layer	scalability layer index
nch	the number of channel
layer_group[layer]	indicates the group index of the spectral data to be added newly in the scalability layer. See subclause 4.5.2.6.2.5
layer_start_sfb[layer]	indicates the index of the lowest scalefactor band index to be added newly in the scalability layer. See subclause 4.5.2.6.2.5
layer_end_sfb[layer]	indicates the highest scalefactor band index to be added newly in the scalability layer. See subclause 4.5.2.6.2.5
scf[ch][g][sfb]	indicates the scalefactors.
max _noise_energy[ch]	indicates the maximum of the noise energy.
dpcm_noise_energy_index[ch][g][sfb]	indicates the differential noise energy index.
is_position_index[g][sfb]	indicates the intensity stereo poistion index.

4.6.4.4.3 Decoding process

The spectral coefficients are divided into scalefactor bands that contain a multiple of 4 quantized spectral coefficients. Each scalefactor band has a scalefactor.

The differential scalefactor index is arithmetic-decoded using the arithmetic model given in Table 4.A.30. The arithmetic model of the scalefactor for the base layer is given as a 3 bit unsigned integer data element, **base_scf_model**. The arithmetic model of the scalefactor for the enhancement layers is given as a 3 bit unsigned integer data element, **enh_scf_model**.

For all scalefactors the difference to the offset value is arithmetic-decoded. All scalefactors are calculated from the difference and the offset value. The offset value is given explicitly as a 8 bit PCM in the data element **max_scalefactor[ch]**. Detailed arithmetic decoding procedure is described in this subclause 4.5.2.6.2.7.

The following pseudo code describes how to decode the scalefactors scf[ch][g][sfb] *in base layer and each enhancement layer*:

```
for (ch = 0; ch < nch; ch++) {
   g = layer_group[layer];
   for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++) {
      diff_scf = arithmetic_decoding();
      scf[ch][g][sfb] = max_scalefactor[ch] – diff_scf;
   }
}
```

If noise substituition coding is active for a particular group and scalefactor band, a noise energy value is transmitted instead of the scalefactor of the respective channel.

Noise energies are arithmetic-coded of differential values. For all noise energies the difference to the offset value is arithmetic-decoded. All noise energies are calculated from the difference and the offset value. The offset value, max_noise_energy[ch] is arithmetic-decoded before the first differential noise energy is decoded.

Noise substitution decoding process is same as the PNS part of MPEG-4 Audio General Audio. The noise energy decoding in each layer is defined by the following pseudo code:

```
for (ch =0; ch < nch; ch++) {
    g = layer_group[layer];
    for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++) {
        if (noise_flag[ch][g][sfb]) {
            dpcm_noise_energy_index[ch][g][sfb] = arithmetic_decoding();
            noise_nrg[ch][g][sfb] = max_noise_energy[ch] - dpcm_noise_energy[ch][g][sfb];
        }
    }
}
```

The direction information for the intensity stereo decoding is represented by an “intensity stereo position” value indicating the relation between left and right channel scaling. If intensity stereois active for a particular group and scalefactor band, an intensity stereo position value is transmitted in stead of the scalefactor of the right channel.

When intensity positions are arithmetic-coded, the same arithmetic model is used. The intensity decoding process is same as the intensity stereo of MPEG-4 Audio General Audio. The intensity stereo position decoding in each layer is defined by the following pseudo code:

```
g = layer_group[layer]
for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++) {
    if (stereo_info[g][sfb] && ch==1) {
        is_position_index[g][sfb] = arithmetic_decoding();
        if (is_position_sign[g][sfb]%2)
            is_position[g][sfb] = -(int)((is_position_index[g][sfb]+1)/2);
        else
            is_position[g][sfb] = (int)(is_position_index[g][sfb]/2);
    }
}
```

4.6.4.5 Decoding of coding band side information

4.6.4.5.1 Description

In BSAC scalable coding scheme, the spectral coefficients are divided into coding bands which contain 32 quantized spectral coefficients for the noiseless coding. Coding bands are the basic units used for the noiseless coding. The set of bit-sliced sequence is divided into coding bands. The MSB plane and the probability table of each coding band are included in this layer coding band side information, **cband_si** as shown in Table 4.A.31. The coding band side informations of each layer are transmitted starting from the lowest coding band (layer_start_cband[layer]) and progressing to the highest coding band (layer_end_cband[layer]). For all cband_si, it is arithmetic-coded using the arithmetic model as given in Table 4.A.29**.**

4.6.4.5.2 Definitions

Data element:

acode_cband_si[ch][g][cband]	Arithmetic codeword from the arithmetic coding of **cband_si** for each coding-band.

Help elements:

g	group index
cband	coding band index within group
ch	channel index
nch	the number of channel
layer_group[layer]	indicates the group index of the spectral data to be added newly in the scalability layer. See subclause 4.5.2.6.2.5

layer_start_cband[layer]	indicates the lowest coding band index to be added newly in the scalability layer. See subclause 4.5.2.6.2.5
layer_end_cband[layer]	indicates the highest coding band index to be added newly in the scalability layer. See subclause 4.5.2.6.2.5

4.6.4.5.3 Decoding process

cband_si is arithmetic-coded using the arithmetic model as given in Table 4.A.29. The arithmetic model used for coding cband_si is dependent on a 5-bit unsigned integer in the data element, **cband_si_type** as shown in Table 4.A.29. And, the largest value of the decodable cband_si is given in Table 4.A.29. If the decoded cband_si larger than this value, it can be considered that there was a bit-error in the bitream.Detailed arithmetic decoding procedure is described in this subclause 4.5.2.6.2.7.

The following pseudo code describes how to decode the cband_si cband_si[ch][g][cband] *in base layer and each enhancement layer*:

```
g= layer_group[layer];
for (ch=0; ch<nch;ch++) {
    for (cband=layer_start_cband[layer]; cband<layer_end_cband [layer]; cband++) {
       cband_si[ch][g][cband] = arithmetic_decoding();
       if (cband_si[ch][g][cband] > largest_cband_si)
          bit_error_is_generated;
   }
}
```

where, layer_start_cband [layer] is the start coding band and layer_end_cband[layer] is the end coding band for decoding the arithmetic model index in each layer.

4.6.4.6 Segmented binary arihmetic coding (SBA)

4.6.4.6.1 Tool Description

Segmented Binary Arithmetic Coding (SBA) is based on the fact that the arithmetic codewords can be partitioned at known positions so that these codewords can be decoded independent of any error within other sections. Therefore, this tool avoids error propagation to those sections. The arithmetic coding should initialized at the beginning of these segments and terminated at the end of these segments in order to localize the arithmetic codewords. This tool is activated if the syntax element, **sba_mode** is 1. And this flag should be set to 1 if the BSAC is used in the error-prone environment.

4.6.4.6.2 Definitions

There is no definition because only the initialization and termination process are added at the beginning and the end of the segments in order to localize the arithmetic codewords.

4.6.4.6.3 Decoding process

The arithmetic coding is terminated at the end of the segments, and re-initialized at the beginning of the next segment. The segment is made up of the scalability layers. terminal_layer[layer] indicates whether each layer is the last layer of the segment, which is set as follows :

```
for (layer = 0; layer < (top_layer+slayer_size-1); layer++) {
   if (layer_start_cband[layer] != layer_start_cband[layer+1])
      terminal_layer[layer] = 1;
   else
      terminal_layer[layer] = 0;
   }
}
```

where, toplayer is the top layer to be encoded, layer_max_cband[] are the maximum coding band limit to be encoded and slayer_size is the sub-layer size of the base layer. Figure 4.26 shows an example of the segmented bistream to be made in the encoder.

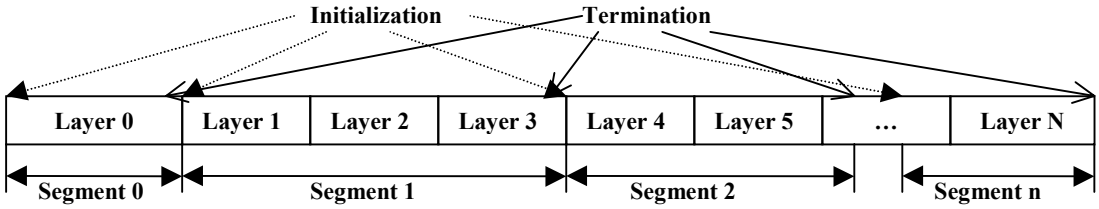


Figure 4.26 – The structure of SBA coded bitstream payload

In the decoder, the bitstream payload of each layer is split from the total bitstream payload. If the previous layer is the last of the segment, the split bitstream payload is stored in the independent buffer and arithmetic decoding process is re-initialized. Otherwise, the split bitstream payload is concatenated to that of the previous layer and used for arithmetic decoding sequentially.

In order to do the arithmetic decoding perfectly, 32-bit zero value should be concatenated to the split bitstream payload if the layer is the last of the segment. Figure 4.27 shows an example of the bitstream payload spliting and zero stuffing in decoder part.

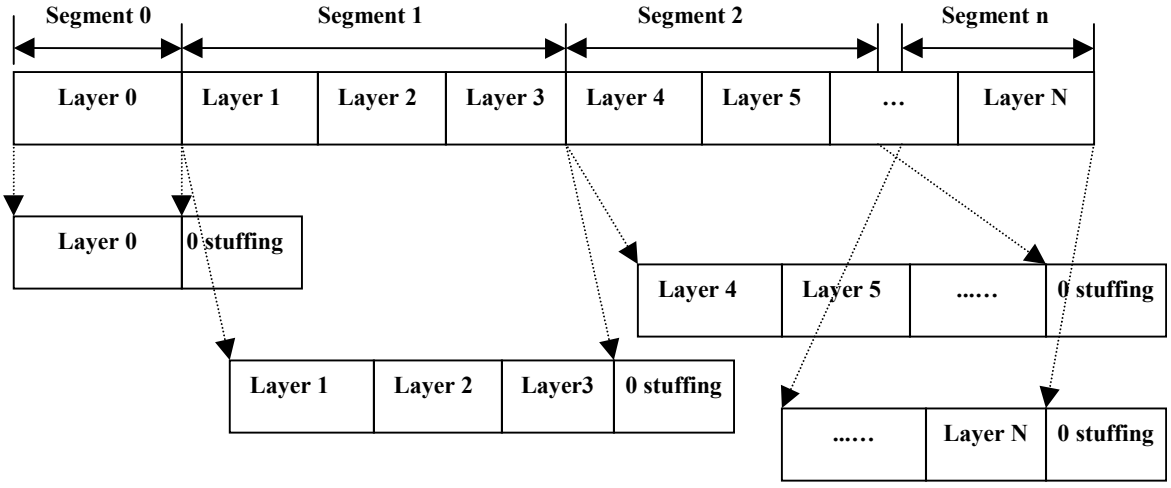


Figure 4.27 – Decoding of SBA bitstream payload

4.6.5 Interleaved vector quantization

4.6.5.1 Tool description

This process generates flattened MDCT spectrum using vector quantization. This quantization tool provides high coding gain, even at lower bitrates. Bitstream payload for this quantizer has a simple fixed-length structure, thus it is robust against transmission channel errors.

The decoding process consists of vector quantization part and reconstruction part. In the vector quantization part, subvectors are specified by codevector index. Then, subvectors are interleaved and combined into one output vector (see Figure 4.28).

4.6.5.2 Definitions

Inputs:

fb_shift[][] Syntax element indicating the active frequency band of the adaptive bandwidth control.

index0[] data element indicating the codevector number of codebook 0

index1[]	data element indicating the codevector number of codebook 1
window_sequence	data element indicating window sequence type
side_info_bits	number of bits for side information
bitrate	system parameter indicating bitrate
used_bits	number of bits used by variable bit-rate tool, such as long term prediction tool
lyr	indicates enhancement layer number. Number 0 is assigned for the base layer.

Outputs:

x_flat[]	reconstructed MDCT coefficients

Parameters:

FRAME_SIZE	frame length
MAXBIT	maximum bits for shape codebook index representation
N_CH	number of channels
N_DIV	number of subvectors
N_SF	number of subframes in a frame
sp_cv0[][]	shape codebook of conjugate channel 0 (Elements are given in Table 4.A.19, Table 4.A.21, Table 4.A.23, Table 4.A.25)
sp_cv1[][]	shape codebook of conjugate channel 1 (Elements are given in tables Table 4.A.20, Table 4.A.22, Table 4.A.24, Table 4.A.26)
SP_CB_SIZE	shape codebook size
shape_index0	points the selected codevector of MDCT shape codebook 0
shape_index1	points the selected codevector of MDCT shape codebook 1
pol0	negates the selected codevector of MDCT shape codebook 0
pol1	negates the selected codevector of MDCT shape codebook 1

4.6.5.3 Parameter settings

The shape codebook vectors of the flattened MDCT coefficients, sp_cv0[][] and sp_cv1[][] are listed in Annex 4.A.

Parameters are set initially as listed below:

MAXBIT_SHAPE = 5

MAXBIT = MAXBIT_SHAPE + 1

SP_CB_SIZE = (1<<MAXBIT_SHAPE)

4.6.5.4 Decoding process

4.6.5.4.1 Initializations

Based on the bits_available_vq defined in 4.5.2.5.3.2,

N_DIV and the length of each subvector, length[], are calculated by

N_DIV and the length of each subvector, length[] is calculated by

```
N_DIV = ((int)((bits_available_vq + MAXBIT*2-1)/(MAXBIT*2)))
```

```
for(idiv=0; idiv<ntt_N_DIV; idiv++){
    length[idiv] = ((int) (N_FR*qsample)*N_SF*N_CH+N_DIV-1-idiv) / N_DIV;
}
```

where N_FR is the number of samples in a subframe and qsample is defined in subclause 4.6.5.4.4

Detailed calcuration of side_info_bits is described in subclause 4.5.2.5.3 with bits_for_side_information.

If the codevector length, length[], exceeds the number of codevector elements which are described in subclause 4.A.4, the undefined elements of sp_cv0[] and sp_cv1[] (i.e. elements beyond the defined area) are set to zero.

4.6.5.4.2 Index unpacking

The quantization index consists of the polarity and shape code information. So in the first stage of the inverse quantization, the input indices are unpacked, and the polarities and shapes are extracted.

The extracting of the polarities is described as follows:

```
for (idiv = 0; idiv < N_DIV; idiv++){
    pol0[idiv] = 2 * (index0 [idiv] / SP_CB_SIZE) - 1;
    pol1[idiv] = 2 * (index1 [idiv] / SP_CB_SIZE) - 1;
}
```

where

pol0[]: polarity of conjugate channel 0
pol1[]: polarity of conjugate channel 1

The shape code extraction is described as follows:

```
for (idiv = 0; idiv < N_DIV; idiv++){
    index_shape0[idiv] = index0 [idiv] % SP_CB_SIZE;
    index_shape1[idiv] = index1 [idiv] % SP_CB_SIZE;
}
```

4.6.5.4.3 Reconstruction

Output coefficients are reconstructed as follows:

```
for (idiv = 0; idiv <  N_DIV; idiv++) {
   for (icv = 0; icv < length[idiv]; icv++) {
      if ((icv<length[0]-1) &&
          ((N_DIV%(N_SF*N_CH)==0 && (N_SF*N_CH)>1) || ((N_SF*N_CH)&0x1)==0)))
         itmp = ((idiv+icv)%N_DIV)+icv*N_DIV;
      else
         itmp = idiv + icv * N_DIV;
      ismp = itmp / (N_SF*N_CH) + ((itmp % (N_SF*N_CH)) * (FRAME_SIZE / (N_SF*N_CH)));
      x_flat_tmp[ismp] =
         (pol0[idiv]*sp_cv0[index_shape0[idiv]][icv]
          + pol1[idiv]*sp_cv1[index_shape1[idiv]][icv]) / 2;
   }
}
```

where
icv: indicates sample number in the shape code vector
idiv: indicates interleaved-division subvector

ismp: indicates sample number in the subframe
itmp: an integer

4.6.5.4.4 Adaptive active band selection

This procedure selects the active band (see Figure 4.29) depending on the **fb_shift** parameter.

If lyr=0, the active band is fixed as listed below:

```
bandUpper_i = 95 * BPS/ ISAMPF;
bandUpper_i = min(100000, bandUpper_i);
bandUpper_i *= 16384;
bandUpper_i += 1562
bandUpper_i /= 3125;
qsample = (double)(bandupper_i)/524288.;
AC_TOP[lyr][i_ch][0] = qsample;
AC_BTM[lyr][i_ch][0] = 0.0;
```

where ISAMP is an integer sampling frequency in [kHz] truncated from the standard frequency values listed in the right column of Table 4.68 and

BPS is bitrate in [bits/s/ch] based on the byte-aligned bits for a frame and it equals

(int)(((FRAME_SIZE * bitrate/sampling_frequency)/8+0.5)*8)*sampling_frequency/FRAME_SIZE/N_CH.

If lyr is greater or equal '1', the upper limit of the quantization bandwidth of a certain layer is defined as follows using the accumulated bitrate totalbps up to the layer.

```
totalbps=bpsbase;
for (iscl0 = 1; iscl0 <= lyr; iscl0++){
   totalbps += BPS_SCL[iscl0];
}

    upperlimit_i = (totalbps* 100) / ISAMPF;
    upperlimit_i = min(100000, upperlimit_i);
    upperlimit_i *= 16384;
    upperlimit_i += 1562;
    upperlimit_i /= 3125;
    upperlimit = (double)(upperlimit_i)/524288.;
```

UPPER_BOUNDARY and LOWER_BOUNDARY of the quantization bandwidth is defined in the tables of AC_TOP[lyr][i_ch][**fb_shift**] and AC_BTM[lyr][i_ch][**fb_shift**] as follows, depending on the band selection code and the value of bitrate per channel BPS_SCL[lyr] in [bits/s/ch] for each enhancement layer.

Note that BPS_SCL[lyr] should be based on the byte-aligned bits for a frame and it equals

(int)(((FRAME_SIZE * bitrate/sampling_frequency)/8+0.5)*8)*sampling_frequency/FRAME_SIZE/N_CH.

```
 qsample_i = (BPS_SCL[lyr]* 130) / ISAMPF;
 qsample_i = min(100000, qsample_i);
 qsample_i *= 16384;
 qsample_i += 1562;
 qsample_i /= 3125;
 bias_i = (upperlimit_i- qsample_i)/4;
 if (qsample_i < bias_i) {
    bias_i = upperlimit_i/4;
   qsample_i = bias_i;
}
 bias = (double) bias_i;
qsample = (double)(qsample_i)/524288./* 16384*32 */
if (bias <= 0.0) bias =0.0;
 for (i_ch = 0; i_ch < N_CH; i_ch++) {
    AC_TOP[lyr][i_ch][0] = qsample;
    AC_BTM[lyr][i_ch][0] = 0.0;
    AC_TOP[lyr][i_ch][1] = qsample+bias;
```

```
        AC_BTM[lyr][i_ch][1] = bias;
        AC_TOP[lyr][i_ch][2] = qsample + bias*2;
        AC_BTM[lyr][i_ch][2] = bias*2;
        AC_TOP[lyr][i_ch][3] = qsample + bias*3;
        AC_BTM[lyr][i_ch][3] = bias*3;
      }
```

where AC_BTM and AC_TOP is the bottom and top frequency of the active band, respectively. Values are ranged from 0 to 1.0 (i.e. 1.0 means the highest frequency).

The lower and upper boundaries in MDCT domain are calculated as follows:

```
for (i_ch = 0; i_ch < N_CH; i_ch++){
      LO WER_BOUNDARY[lyr][i_ch] = AC_BTM[lyr][i_ch][fb_shft] * N_FR;

      UPPER_BOUNDARY[lyr][i_ch] = AC_TOP[lyr][i_ch][fb_shift] * N_FR;

}
```

Then, the output x_flat[] is copied from x_flat_tmp[] as follows:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   for (isf = 0; isf < N_SF; isf++) {
      for (ismp = 0; ismp < LOWER_BOUNDARY[lyr][i_ch]; ismp++) {
         x_flat[ismp+(isf+i_ch*N_SF)*N_FR] = 0.;
      }
      for (ismp = LOWER_BOUNDARY[lyr][i_ch]; ismp < UPPER_BOUNDARY[lyr][i_ch]; ismp++) {
         ismp2 = ismp - LOWER_BOUNDARY[lyr][i_ch];
         x_flat[ismp+(isf+i_ch*N_SF)*N_FR] = x_flat_tmp[ismp2+(isf+i_ch*N_SF)*N_FR];
      }
      for (ismp = UPPER_BOUNDARY[lyr][i_ch]; ismp < N_FR; ismp++) {
         x_flat[ismp+(isf+i_ch*N_SF)*N_FR] = 0.;
      }
   }
}
```

4.6.5.5 Diagrams

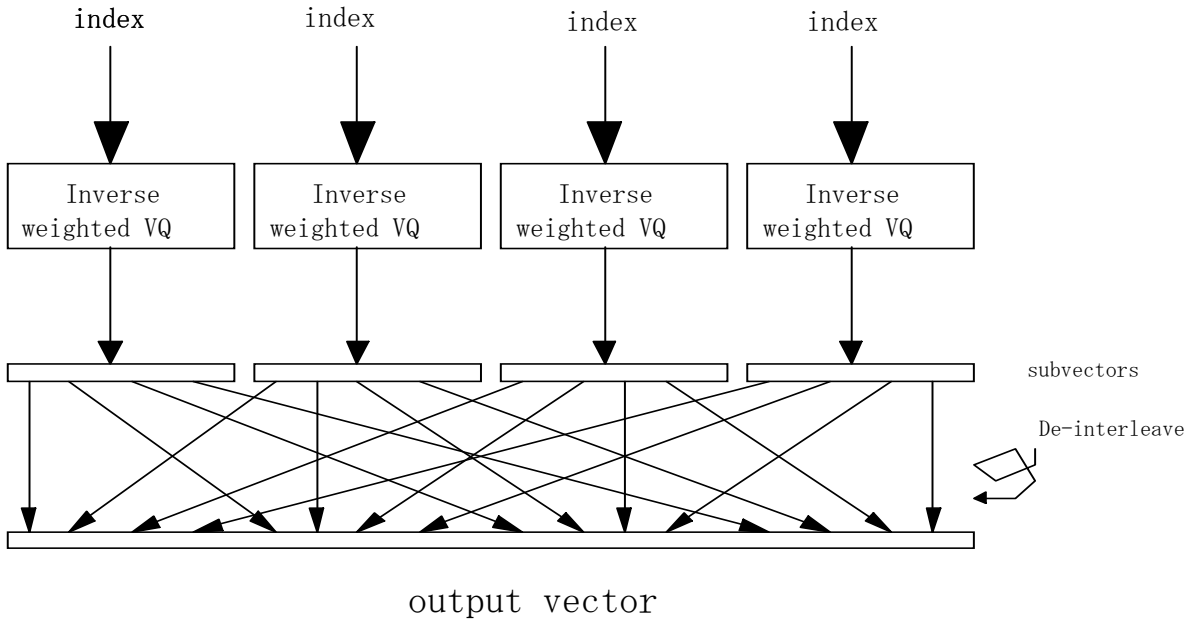


Figure 4.28 – Decoding process of interleaved vector quantization tool.

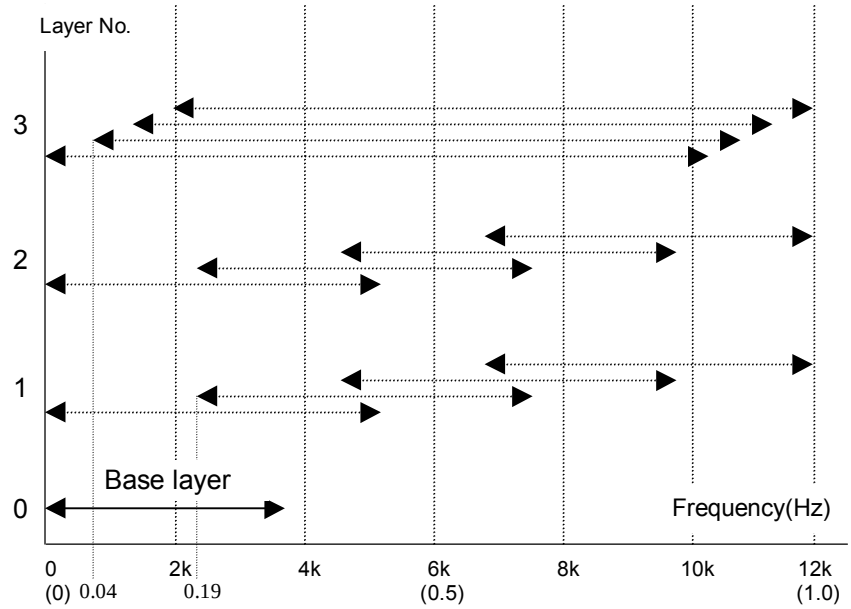


Figure 4.29 – Example of adaptive active band selection.
(Sampling_Frequency=24kHz, Total bitrate=40kbit/s(8+8+8+16kbps)
In this case, qsample = 0.43, bias = 0.19 for layer 1 and layer 2
qsample = 0.87, bias = 0.04 for layer 3)

4.6.6 Frequency domain prediction

See ISO/IEC13818-7 (13818-7:2005, subclause 13.3.2 "Predictor Processing").

Notes:

The use of the prediction tool is object type / profile dependent. See subpart 1 for detailed information on the MPEG-4 Audio object types and profiles.

The frequency domain prediction tool can be used only for AudioObjectType 1 (AAC Main).

4.6.7 Long term prediction (LTP)

4.6.7.1 Tool description

Long term prediction (LTP) is an efficient tool for reducing the redundancy of a signal between successive coding frames. This tool is especially effective for the parts of a signal which have clear pitch property. The implementation complexity of LTP is significantly lower than the complexity of Backward Adaptive Prediction. Because the Long Term Predictor is a forward adaptive predictor (prediction coefficients are sent as side information), it is inherently less sensitive to round-off numerical errors in the decoder or bit errors in the transmitted spectral coefficients.

With the MPEG-4 AAC scalable audio object type LTP can be used for any window type. For the AAC LTP audio object type LTP is restricted to long windows only, to achieve bitstream payload compatibility with MPEG-2 AAC such that if LTP is not used, an ISO/IEC 13818-7 (MPEG-2) AAC LC or Main profile decoder can parse the MPEG-4 AAC LTP bitstream payload.

4.6.7.2 Definitions

ltp_data_present 1 bit indicating whether prediction is used in current frame (1) or not (0) (always present)

ltp_lag 11-bit number specifying the optimal delay from 0 to 2047.

ltp_coef 3-bit index indicating the LTP coefficient in the table below.

Table 4.133 – LTP coefficient

Value of **ltp_coef**	value of LTP coefficient
000	0.570829
001	0.696616

010	0.813004
011	0.911304
100	0.984900
101	1.067894
110	1.194601
111	1.369533

ltp_long_used 1 bit for each scalefactor band (sfb) where LTP can be used indicating whether LTP is switched on (1) or off (0) in that sfb.

For the audio object type ER AAC LD the following data elements are available:

ltp_lag_update 1 bit indicating whether a new value for the ltp lag is transmitted. Due to the high consistency of the LTP lag for many signals, one additional bit is introduced signaling that the lag of the previous frame is repeated (ltp_lag_update == 0). Otherwise, a new value for the ltp lag is transmitted (ltp_lag_update == 1).

ltp_lag 10-bit number specifying the optimal delay from 0 to 1023.

The maximum number of scalefactor bands used in prediction is limited by a constant:

MAX_LTP_LONG_SFB = 40 (for long frames)

4.6.7.3 Decoding process

The decoding process for LTP is carried out on each window of the current frame by applying 1-tap IIR filtering in the time domain to predict samples in the current frame by (quantized) samples in the previous frames. The process is controlled by the transmitted side information in a two step approach. The first control step defines whether LTP is used at all for the current frame. In the case of long window, the second control step defines, on which scalefactor bands LTP is used. At the start of the decoding process, the reconstructed time domain samples are initialized by zeros.

For each frame, the LTP side information is extracted from the bitstream payload to control the further predictor processing at the decoder. In the case of a single_channel_element() the control information is valid for the channel with that element. In the case of a channel_pair_element() there are two sets of control data. The order of occurrence of LTP control data sets in the case of **common_window==1** is such that LTP side information is first extracted for the left channel and then for the right channel of the corresponding channel_pair_element().

First, the **ltp_data_present** bit is read. If this bit is not set (0) then LTP is switched off for the current frame and there is no further predictor side information present. In this case the **ltp_long_used** flag for each scalefactor band stored in the decoder has to be set to zero. If the **ltp_data_present** bit is set (1) then LTP is used for the current frame and the LTP parameters are read.

For long windows, the LTP parameters are used to calculate the predicted time domain signals using the following formula:

$x\_est(i) = ltp\_coef * x\_rec(i - M - ltp\_lag)$

$i = 0,\ldots,N-1$

where $x\_est(i)$ are the predicted samples

$x\_rec(i)$ are reconstructed time domain samples

N is the length of the transform window

M = N/2 if aot == ER AAC LD, otherwise M = 0

The different value for M used in combination with the low delay codec is to achieve a similar range of possible lag values in absolute time, despite of the shorter frame.

The reference point for index i and the content of the buffer x_rec are arranged so that x_rec(0 ... N/2 - 1) contains the last aliased half window from the IMDCT, and x_rec(N/2 ... N-1) is always all zeros. The rest of x_rec (i<0) contains the previous fully reconstructed time domain samples, i.e., output of the decoder. To reduce memory consumption the samples in the LTP-buffer (x_rec) are stored as 16-bit integer numbers. These rounded 16-bit integer numbers are used to calculate the predicted time domain signal. The exact rounding algorithm to be used is shown in the following pseudo-C function LTP_Round().

```
static short
LTP_Round(double x_rec)
{
  short out_value;

  if(x_rec > 0x7FFF)
    out_value = 0x7FFF;
  else if(x_rec < -0x8000)
   out_value = -0x8000;
  else
  {
    out_value = INT(|x_rec| + 0.5f);
    if(x_rec  < 0.0f) out_value = -out_value;
  }

  return (out_value);
}
```

Using the MDCT for long windows, the predicted spectral components are obtained for the current frame from the predicted time domain signal. Next, the **ltp_long_used** bits are read from the bitstream payload, which control the use of prediction in each scalefactor band individually, i.e. if the bit is set for a particular scalefactor band, all the predicted spectral components of this scalefactor band are used. Otherwise the predicted spectral components are set to zeros. That is, if the **ltp_long_used** bit is set, then the quantized prediction error reconstructed from the transmitted data is added to the predicted spectral component. If the bit is not set (0), then the quantized value of spectral component is reconstructed directly from the transmitted data.

The signal reconstruction part of decoding process for one channel can be described as following pseudo code. Here *x_est* is the predicted time domain signal, *X_est* is the corresponding frequency domain vector, *Y_rec* is the vector of decoded spectral coefficients and *X_rec* is the vector of reconstructed spectral coefficients.

```
if (ONLY_LONG_SEQUENCE || LONG_START_SEQUENCE || LONG_STOP_SEQUENCE) {
    x_est = predict();
    X_est = MDCT(x_est);
    for (sfb = 0; sfb < min(max_sfb, MAX_LTP_LONG_SFB); sfb++) {
        if (ltp_data_present && ltp_long_used[sfb])
            X_rec = X_est + Y_rec;
        else
            X_rec = Y_rec;
    }
}
```

4.6.7.4 Integration of LTP with other GA tools

4.6.7.4.1 LTP with TNS

Because TNS needs to be applied to a reconstructed spectrum, the TNS filter must occur after LTP in the decoding chain. This makes an additional TNS analysis filter necessary in the LTP loop, as shown in Figure 4.30.

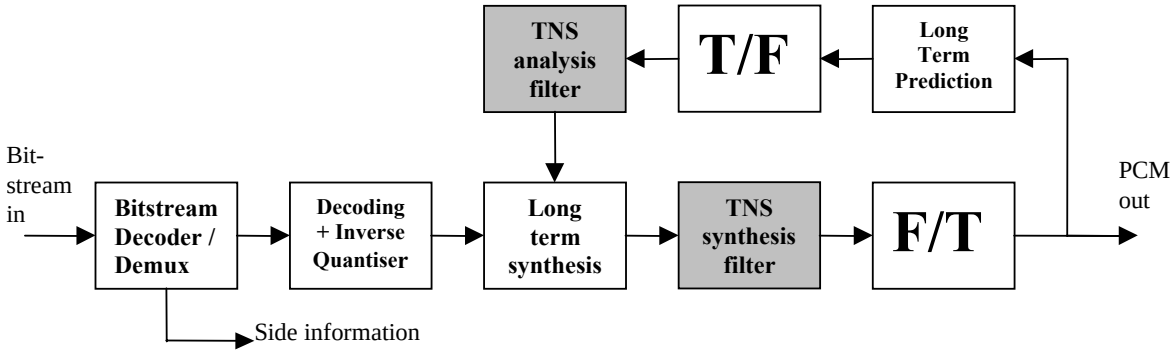


Figure 4.30 – Long Term Prediction with TNS

4.6.7.4.2 LTP with PNS

Simultaneous use of LTP and PNS is not prevented in the syntax. If both LTP, and PNS are enabled on the same scalefactor band, PNS takes precedence, and no prediction is applied to this band.

4.6.7.4.3 LTP with dependently switched coupling

No dependently switched coupling and hence no dependently switched CCE is permitted in any audio object type that utilizes LTP.

4.6.7.5 LTP in a scalable GA decoder

In a scalable coder LTP is used only on the lowest GA coding layer and the predictor update is based on the time domain output of the first GA layer. Either LTP or a core coder can be used, but not both at the same time. The LTP decoding process is similar to the process used when the lowest layer is a core coder. The LTP part itself is decoded exactly in the same way as in a non-scalable GA decoder.

In a scalable configuration the simultaneous use of LTP and Intensity Stereo is not prevented in the syntax. However if both LTP and Intensity Stereo are enabled in the same scalefactor band in the first GA layer, Intensity Stereo takes precedence and no prediction is applied to this band.

4.6.8 *Joint Coding*

(similar to ISO/IEC 13818-7)

4.6.8.1 M/S stereo

4.6.8.1.1 Tool description

The M/S joint channel coding operates on channel pairs. Channels are most often paired such that they have symmetric presentation relative to the listener, such as left/right or left surround/right surround. The first channel in the pair is denoted „left“ and the second „right.“ On a per-spectral-coefficient basis, the vector formed by the left and right channel signals is reconstructed or de-matrixed by either the identity matrix

$$\begin{bmatrix} l \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} l \\ r \end{bmatrix}$$

or the inverse M/S matrix

$$\begin{bmatrix} l \\ r \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} m \\ s \end{bmatrix}$$

The decision on which matrix to use is done on a scalefactor band by scalefactor band basis as indicated by the ms_used flags. For the audio object types AAC Main, LC, SSR, and LTP, M/S joint channel coding can only be used if **common_window** is ‘1’. For the AAC scalable audio object type always common windows between the two audio channels are used, so that M/S coding is always possible.

4.6.8.1.2 Definitions

ms_mask_present	this two bit field indicates that the MS mask is 00 All zeros 01 A mask of max_sfb bands of ms_used follows this field 10 All ones 11 Reserved
ms_used[g][sfb]	one-bit flag per scalefactor band indicating that M/S coding is being used in windowgroup g and scalefactor band sfb.
l_spec[]	Array containing the left channel spectrum of the respective channel pair.
r_spec[]	Array containing the right channel spectrum of the respective channel pair.
is_intensity(g,sfb)	function returning the intensity status, defined in subclause 4.6.8.2.3.
is_noise(g,sfb)	function returning the noise substitution status, defined in subclause 4.6.13.3.

4.6.8.1.3 Decoding process

Reconstruct the spectral coefficients of the first („left“) and second („right“) channel as specified by the **ms_mask_present** and the **ms_used[][]** flags as follows:

```
if (ms_mask_present >= 1) {
   for (g = 0; g < num_window_groups; g++) {
      for (b = 0; b < window_group_length[g]; b++) {
         for (sfb = 0; sfb < max_sfb; sfb++) {
            if  ((ms_used[g][sfb] || ms_mask_present == 2) &&
                  !is_intensity(g,sfb) && !is_noise(g,sfb)) {
               for (i = 0; i< swb_offset[sfb+1]-swb_offset[sfb]; i++) {
                  tmp = l_spec[g][b][sfb][i] -  r_spec[g][b][sfb][i];
                  l_spec[g][b][sfb][i] =     l_spec[g][b][sfb][i] +
                                             r_spec[g][b][sfb][i];
                  r_spec[g][b][sfb][i] = tmp;
               }
            }
         }
      }
   }
}
```

Please note that ms_used[][] is also used in the context of intensity stereo coding and perceptual noise substitution. If intensity stereo coding or noise substitution is on for a particular scalefactor band, no M/S stereo decoding is carried out.

4.6.8.1.4 Integration of the M/S stereo tool for the audio object type AAC scalable

The same MS mask is applied to all layers. If subsequent layers specify an increasing max_sfb, ms_mask_present and ms_used[][] are transmitted for the additional scale factor bands and groups only (see Table 4.54 - Syntax of ms_data()).

4.6.8.2 Intensity Stereo (IS)

4.6.8.2.1 Tool description

This tool is used to implement joint intensity stereo coding between both channels of a channel pair. Thus, both channel outputs are derived from a single set of spectral coefficients after the inverse quantization process. This is done selectively on a scalefactor band basis when intensity stereo is flagged as active.

4.6.8.2.2 Definitions

hcod_sf[]	Huffman codeword from the Huffman code table used for coding of scalefactors (see subclause 4.6.3.2)
dpcm_is_position[][]	Differentially encoded intensity stereo position
is_position[group][sfb]	Intensity stereo position for each group and scalefactor band
l_spec[]	Array containing the left channel spectrum of the respective channel pair

r_spec[] Array containing the right channel spectrum of the respective channel pair

4.6.8.2.3 Decoding process

The use of intensity stereo coding is signaled by the use of the pseudo codebooks INTENSITY_HCB and INTENSITY_HCB2 (15 and 14) only in the right channel of a channel_pair_element() having a common ics_info() (**common_window** == 1). INTENSITY_HCB and INTENSITY_HCB2 signal in-phase and out-of-phase intensity stereo coding, respectively.

In addition, in case of a non-scalable GA decoder the phase relationship of the intensity stereo coding can be reversed by means of the ms_used field: Because M/S stereo coding and intensity stereo coding are mutually exclusive for a particular scalefactor band and group, the primary phase relationship indicated by the Huffman code tables is changed from in-phase to out-of-phase or vice versa if the corresponding ms_used bit is set for the respective band.

The directional information for the intensity stereo decoding is represented by an "intensity stereo position" value indicating the relation between left and right channel scaling. If intensity stereo coding is active for a particular group and scalefactor band, an intensity stereo position value is transmitted instead of the scalefactor of the right channel. Intensity positions are coded just like scalefactors, i.e. by Huffman coding of differential values with two differences:

- There is no first value that is sent as PCM. Instead, the differential decoding is started assuming the last intensity stereo position value to be zero.
- Differential decoding is done separately between scalefactors and intensity stereo positions. In other words, the scalefactor decoder ignores interposed intensity stereo position values and vice versa (see subclause 4.6.2.3.2)

The same codebook is used for coding intensity stereo positions as for scalefactors.

Two pseudo functions are defined for use in intensity stereo decoding:

```
function is_intensity(group,sfb) {
   +1     for window groups / scalefactor bands with right channel
             codebook sfb_cb[group][sfb] == INTENSITY_HCB
   -1     for window groups / scalefactor bands with right channel
             codebook sfb_cb[group][sfb] == INTENSITY_HCB2
   0      otherwise
}

function invert_intensity(group,sfb) {
   1-2*ms_used[group][sfb]    if (ms_mask_present == 1) && aot != AAC scalable
   +1                         otherwise
}
```

The intensity stereo decoding for one channel pair is defined by the following pseudo code:

```
p = 0;
for (g = 0; g < num_window_groups; g++)  {

   /* Decode intensity positions for this group */
   for (sfb = 0; sfb < max_sfb; sfb++)
      if (is_intensity(g,sfb))
         is_position[g][sfb] = p += dpcm_is_position[g][sfb];

   /* Do intensity stereo decoding */
   for (b = 0; b < window_group_length[g]; b++)  {
      for (sfb = 0; sfb < max_sfb; sfb++)  {
         if (is_intensity(g,sfb))  {

            scale = is_intensity(g,sfb) * invert_intensity(g,sfb) *
                        0.5^(0.25*is_position[g][sfb]);
            /* Scale from left to right channel,
               do not touch left channel */
            for (i = 0; i < swb_offset[sfb+1]-swb_offset[sfb]; i++)
               r_spec[g][b][sfb][i] = scale * l_spec[g][b][sfb][i];
         }
      }
   }
}
```

In case of reversible variable length coding (RVLC) there is no last value that is sent as PCM. Instead, the backwards differential decoding is started assuming the last intensity stereo position value to be zero. The decoding of the intensity stereo positions is defined by the following pseudo code:

```
p = dpcm_is_last_position;

for (g = win-1; g >= 0; g--) {
  for (sfb = sfbmax-1; sfb >= 0; sfb--) {
     is_pos[g][sfb]=p;
      p -= dpcm_is_pos[g][sfb];
  }
}
```

4.6.8.2.4 Integration with the intra channel prediction tool

For scalefactor bands coded in intensity stereo the corresponding predictors in the right channel are switched to "off" thus effectively overriding the status specified by the prediction_used mask. The update of these predictors is done by feeding the intensity stereo decoded spectral values of the right channel as the "last quantized value" $x_{rec(n-1)}$. These values result from the scaling process from left to right channel as described in the pseudo code.

4.6.8.2.5 Integration with the long term prediction tool

In case of a non-scalable configuration the function of Long Term Prediction does not depend on Intensity Stereo. The LTP tool shall be applied to both channels of a channel pair element after intensity stereo coding has been carried out.

4.6.8.2.6 Integration of the intensity tool for the audio object type AAC scalable

If a particular scalefactor band and group is coded by intensity stereo, its contribution to the spectral components of the output signal is omitted if spectral coefficients are transmitted for this scalefactor band and group in any of the higher (enhancement) layers (that contributes to the output signal) by means of a non-intensity codebook number.

If a particular scalefactor band and group is coded by intensity stereo, and if the same scalefactor band in the next enhancement layer is also coded by intensity stereo, only the spectral components of the left channel are combined. The spectral components of the right channel are re-calculated for the enhancement layer using the intensity factor of this enhancement layer.

In a scalable configuration the simultaneous use of Intensity Stereo and LTP is not prevented in the syntax. However if both Intensity Stereo and LTP are enabled in the same scalefactor band in the first GA layer, Intensity Stereo takes precedence and no prediction is applied to this band.

In case of an AAC scalable configuration the ms_used field is ignored in Intensity Stereo decoded scalefactor bands but may still signal the use of M/S stereo decoding in higher (enhancement) layers.

Additional information on decoding intensity tool in a scalable bitstream payload is given in subclause 4.5.2.2.7.

4.6.8.3 Coupling channel

(identical to ISO/IEC 13818-7)

4.6.8.3.1 Tool description

coupling_channel element()'s provide two functionalities: First, coupling channels may be used to implement generalized intensity stereo coding where channel spectra can be shared across channel boundaries. Second, coupling channels may be used to dynamically perform a downmix of one sound object into the stereo image.

Note that this tool includes certain object type dependent parameters (see subclause 4.6.9).

4.6.8.3.2 Definitions

ind_sw_cce_flag — one bit indicating whether the coupled target syntax element is an independently switched (1) or a dependently switched (0).

num_coupled_elements — Number of coupled target channels is equal to num_coupled_elements+1. The minimum value is 0 indicating 1 coupled target channel.

cc_target_is_cpe	one bit indicating if the coupled target syntax element is a CPE (1) or a SCE (0).
cc_target_tag_select	four bit field specifying the element_instance_tag of the coupled target syntax element.
cc_l	one bit indicating that a list of gain_element values is applied to the left channel of a channel pair.
cc_r	one bit indicating that a list of gain_element values is applied to the right channel of a hannel pair).
cc_domain	one bit indicating whether the coupling is performed before (0) or after (1) the TNS decoding of the coupled target channels
gain_element_sign	one bit indicating if the transmitted gain_element values contain information about in-phase / out-of-phase coupling (1) or not (0)
gain_element_scale	determines the amplitude resolution *cc_scale* of the scaling operation according to Table 4.135.
common_gain_element_present[c]	one bit indicating whether Huffman coded common_gain_element values are transmitted (1) or whether Huffman coded differential gain_elements are sent (0)
dpcm_gain_element[][]	Differentially encoded gain element
gain_element[group][sfb]	Gain element for each group and scalefactor band
common_gain_element[]	Gain element that is used for all window groups and scalefactor bands of one coupling target channel
spectrum_m(idx, domain)	Pointer to the spectral data associated with the single_channel_element() with index idx. Depending on the value of "domain", the spectral coefficients before (0) or after (1) TNS decoding are pointed to.
spectrum_l(idx, domain)	Pointer to the spectral data associated with the left channel of the channel_pair_element() with index idx. Depending on the value of "domain", the spectral coefficients before (0) or after (1) TNS decoding are pointed to.
spectrum_r(idx, domain)	Pointer to the spectral data associated with the right channel of the channel_pair_element() with index idx. Depending on the value of "domain", the spectral coefficients before (0) or after (1) TNS decoding are pointed to.

4.6.8.3.3 Decoding process

The coupling channel is based on an embedded single_channel_element() which is combined with some dedicated fields to accomodate its special purpose.

The coupled target syntax elements (SCEs or CPEs) are addressed using two syntax elements. First, the cc_target_is_cpe field selects whether a SCE or CPE is addressed. Second, a cc_target_tag_select filed selects the instance_tag of the SCE/CPE.

The scaling operation involved in channel coupling is defined by gain_element values which describe the applicable gain factor and sign. In accordance with the coding procedures for scalefactors and intensity stereo positions, gain_element values are differentially encoded using the Huffman table for scalefactors. Similarly, the decoded gain factors for coupling relate to window groups of spectral coefficients.

Independently switched CCEs vs. dependently switched CCEs

There are two kinds of CCEs. They are „independently switched“ and „dependently switched“ CCEs. An independently switched CCE is a CCE in which the window state (i.e. window_sequence and window_shape) of the CCE does not have to match that of any of the SCE or CPE channels that the CCE is coupled onto (target channels). This has several important implications:

- First, it is required that an independently switched CCE must only use the common_gain element, not a list of gain_elements.

- Second, the independently switched CCE must be decoded all the way to the time domain (i.e. including the synthesis filterbank) before it is scaled and added onto the various SCE and CPE channels that it is coupled to in the case that window state does not match.

A dependently switched CCE, on the other hand, must have a window state that matches all of the target SCE and CPE channels that it is coupled onto as determined by the list of cc_l and cc_r elements. In this case, the CCE only needs to be decoded as far as the frequency domain and then scaled as directed by the gain list before it is added to the target SCE or CPE channels.

The following pseudo code in function decode_ coupling_channel() defines the decoding operation for a dependently switched coupling channel element. First the spectral coefficients of the embedded single_channel_element() are decoded into an internal buffer. Since the gain elements for the first coupled target (list_index == 0) are not transmitted, all gain_element values associated with this target are assumed to be 0, i.e. the coupling channel is added to the coupled target channel in its natural scaling. Otherwise the spectral coefficients are scaled and added to the coefficients of the coupled target channels using the appropriate list of gain_element values.

An independently switched CCE is decoded like a dependently switched CCE having only common_gain_elements. However, the resulting scaled spectrum is transformed back into its time representation and then coupled in the time domain.

Please note that the gain_element lists may be shared between the left and the right channel of a target channel pair element. This is signalled by both cc_l and cc_r being zero as indicated in the table below:

Table 4.134 – Shared gain_element lists

cc_l, cc_r	shared gain list present	left gain list present	right gain list present
0, 0	yes	no	no
0, 1	no	no	yes
1, 0	no	yes	no
1, 1	no	yes	yes

```
decode_coupling_channel()
{
    - decode spectral coefficients of embedded single_channel_element
      into buffer "cc_spectrum[]".

    /* Couple spectral coefficients onto target channels */
    list_index = 0;
    for (c = 0; c < num_coupled_elements+1; c++)  {
        if (!cc_target_is_cpe[c])  {
            couple_channel( cc_spectrum,
                            spectrum_m( cc_target_tag_select[c], cc_domain ),
                            list_index++ );
        }
        if (cc_target_is_cpe[c])  {
            if (!cc_l[c]  &&  !cc_r[c])  {
                couple_channel( cc_spectrum,
                                spectrum_l( cc_target_tag_select[c], cc_domain ),
                                list_index );
                couple_channel( cc_spectrum,
                                spectrum_r( cc_target_tag_select[c], cc_domain ),
                                list_index++ );
            }
            if (cc_l[c])  {
                couple_channel( cc_spectrum,
                                spectrum_l( cc_target_tag_select[c], cc_domain ),
                                list_index++ ) );
            }
            if (cc_r[c])  {
                couple_channel( cc_spectrum,
                                spectrum_r( cc_target_tag_select[c], cc_domain ),
                                list_index++ ) );
            }
        }
```

```
    }
}

couple_channel( source_spectrum[], dest_spectrum[], gain_list_index )
{
    idx = gain_list_index;
    a = 0;
    cc_scale = cc_scale_table[gain_element_scale];
    for (g = 0; g < num_window_groups; g++)  {

        /* Decode coupling gain elements for this group */
        if (common_gain_element_present[idx])  {

            for (sfb = 0; sfb < max_sfb; sfb++)  {
                cc_sign[idx][g][sfb] = 1;
                gain_element[idx][g][sfb] = common_gain_element[idx];
            }

        }  else  {

            for (sfb = 0; sfb < max_sfb; sfb++)  {
                if (sfb_cb[g][sfb] == ZERO_HCB)
                    continue;

                if (gain_element_sign)  {
                    cc_sign[idx][g][sfb] =
                        1 - 2*(dpcm_gain_element[idx][g][sfb] & 0x1);
                    gain_element[idx][g][sfb] =
                        a += (dpcm_gain_element[idx][g][sfb] >> 1);
                }
                else  {
                    cc_sign[idx][g][sfb] = 1;
                    gain_element[idx][g][sfb] =
                        a += dpcm_gain_element[idx][g][sfb];
                }
            }

        }


        /* Do coupling onto target channels */
        for (b = 0; b < window_group_length[b]; b++)  {
            for (sfb = 0; sfb < max_sfb; sfb++)  {

                if (sfb_cb[g][sfb] != ZERO_HCB)  {
                    cc_gain[idx][g][sfb] =
                        cc_sign[idx][g][sfb] * cc_scale^gain_element[idx][g][sfb];

                    for (i = 0; i < swb_offset[sfb+1]-swb_offset[sfb]; i++)
                        dest_spectrum[g][b][sfb][i] +=
                            cc_gain[idx][g][sfb] * source_spectrum[g][b][sfb][i];

                }
            }
        }
    }
}
```

Note: The array sfb_cb represents the codebook data respect to the CCE's embedded single_channel_element() (not the coupled target channel).

4.6.8.3.4 Tables

Table 4.135 – Scaling resolution for channel coupling (cc_scale_table)

Value of "gain_element_scale"	Amplitude Resolution "cc_scale"	Stepsize [dB]
0	2^(1/8)	0.75

1	2^(1/4)	1.50
2	2^(1/2)	3.00
3	2^1	6.00

Note: the Coupling Channel Tool can only be aplied to AudioObjectTypes AAC Main, LC, SSR, and LTP.

4.6.9 Temporal noise shaping (TNS)

(similar to ISO/IEC 13818-7)

4.6.9.1 Tool description

Temporal Noise Shaping is used to control the temporal shape of the quantization noise within each window of the transform. This is done by applying a filtering process to parts of the spectral data of each channel.

4.6.9.2 Definitions

n_filt[w] number of noise shaping filters used for window w

coef_res[w] token indicating the resolution of the transmitted filter coefficients for window w, switching between a resolution of 3 bits (0) and 4 bits (1)

length[w][filt] length of the region to which one filter is applied in window w (in units of scalefactor bands)

order[w][filt] order of one noise shaping filter applied to window w

direction[w][filt] 1 bit indicating whether the filter is applied in upward (0) or downward (1) direction

coef_compress[w][filt] 1 bit indicating whether the most significant bit of the coefficients of the noise shaping filter filt in window w are omitted from transmission (1) or not (0)

coef[w][filt][i] coefficients of one noise shaping filter applied to window w

spec[w][k] Array containing the spectrum for the window w of the channel being processed

Depending on the window_sequence the size of the following data elements is switched for each transform window according to its window size:

Table 4.136 – Size of data elements

Name	Window with 128 spectral lines	Other window size
'n_filt'	1	2
'length'	4	6
'order'	3	5

4.6.9.3 Decoding process

The decoding process for Temporal Noise Shaping is carried out separately on each window of the current frame by applying all-pole filtering to selected regions of the spectral coefficients (see function tns_decode_frame). The number of noise shaping filters applied to each window is specified by "n_filt". The target range of spectral coefficients is defined in units of scalefactor bands counting down "length" bands from the top band (or the bottom of the previous noise shaping band).

First the transmitted filter coefficients have to be decoded, i.e. conversion to signed numbers, inverse quantization, conversion to LPC coefficients as described in function tns_decode_coef(). Then the all-pole filters are applied to the target frequency regions of the channel's spectral coefficients (see function tns_ar_filter()). The token "direction" is used to determine the direction the filter is slid across the coefficients (0=upward, 1=downward). The constant TNS_MAX_BANDS defines the maximum number of scalefactor bands to which Temporal Noise Shaping is applied. The maximum possible filter order is defined by the constant TNS_MAX_ORDER. Both constants are object type dependent parameters.

The decoding process for one channel can be described as follows pseudo code:

```
/* TNS decoding for one channel and frame */
tns_decode_frame()
{
   for (w = 0; w < num_windows; w++) {

      bottom = num_swb;
      for (f = 0; f < n_filt[w]; f++)  {

         top = bottom;
         bottom = max( top - length[w][f], 0 );
         tns_order = min( order[w][f], TNS_MAX_ORDER );
         if (!tns_order)  continue;

         tns_decode_coef( tns_order, coef_res[w]+3, coef_compress[w][f],
                          coef[w][f], lpc[] );

         start  = swb_offset[min(bottom,TNS_MAX_BANDS,max_sfb)];
         end =   swb_offset[min(top,TNS_MAX_BANDS,max_sfb)];
         if ((size = end - start) <= 0)  continue;

         if (direction[w][f])  {
            inc = -1;   start = end - 1;
         }  else  {
            inc =  1;
         }

         tns_ar_filter( &spec[w][start], size, inc, lpc[], tns_order );

      }

   }
}

/* Decoder transmitted coefficients for one TNS filter */
tns_decode_coef( order, coef_res_bits, coef_compress, coef[], a[] )
{

   /* Some internal tables */
   sgn_mask[]  = {  0x2,  0x4,  0x8 };
   neg_mask[]  = { ~0x3, ~0x7, ~0xf };

                                                    /* size used for transmission */
   coef_res2 = coef_res_bits - coef_compress;
   s_mask = sgn_mask[ coef_res2 - 2 ];  /* mask for sign bit */
   n_mask = neg_mask[ coef_res2 - 2 ];  /* mask for padding neg. values */

   /* Conversion to signed integer */
   for (i = 0; i < order; i++)
      tmp[i] = (coef[i] & s_mask) ? (coef[i] | n_mask) : coef[i];

   /* Inverse quantization */
   iqfac   = ((1 << (coef_res_bits-1)) - 0.5) / (π/2.0);
   iqfac_m = ((1 << (coef_res_bits-1)) + 0.5) / (π/2.0);
   for (i = 0; i < order; i++)  {
      tmp2[i] = sin( tmp[i] / ((tmp[i] >= 0) ? iqfac : iqfac_m) );
   }
}

/* Conversion to LPC coefficients */
a[0] = 1;
for (m = 1; m <= order; m++) {
        for (i = 1; i < m; i++) {           /* loop only while i<m */
                b[i] = a[i] + tmp2[m-1] * a[m-i];
        }
        for (i = 1; i < m; i++) {           /* loop only while i<m */
                a[i] = b[i];
        }
        a[m] = tmp2[m-1];                /* changed */
}
```

```
tns_ar_filter( spectrum[], size, inc, lpc[], order )
{
    - Simple all-pole filter of order "order" defined by
      y(n) =  x(n) - lpc[1]*y(n-1) - ... - lpc[order]*y(n-order)

    - The state variables of the filter are initialized to zero every time

    - The output data is written over the input data ("in-place operation")

    - An input vector of "size" samples is processed and the index increment
      to the next data sample is given by "inc"
}
```

Please note that this pseudo code uses a „C“-style interpretation of arrays and vectors, i.e. if coef[w][filt][i] describes the coefficients for all windows and filters, coef[w][filt] is a pointer to the coefficients of one particular window and filter. Also, the identifier coef is used as a formal parameter in function tns_decode_coef().

4.6.9.4 Maximum TNS order and bandwidth

The value for the constant MAX_TNS_ODER depends on audio object type and windowing, and. Table 4.137 defines MAX_TNS_ORDER depending on these parameters.

Table 4.137 – Definition of TNS_MAX_ORDER depending on AOT and windowing

	windowing	
	short windows	long windows
AOT 1 (AAC Main)	7	20
other AOT using TNS	7	12

According to the sampling rate and audio object type in use, the value for the constant TNS_MAX_BANDS is set as follows:

Table 4.138 – Definition of TNS_MAX_BANDS depending on AOT, windowing and sampling rate

Sampling Rate [Hz]	AudioObject types without PQF filterbank (long windows)	AudioObject types without PQF filterbank (short windows)	AudioObject types with PQF filterbank (long windows)	AudioObject types with PQF filterbank (short windows)
96000	31	9	28	7
88200	31	9	28	7
64000	34	10	27	7
48000	40	14	26	6
44100	42	14	26	6
32000	51	14	26	6
24000	46	14	29	7
22050	46	14	29	7
16000	42	14	23	8
12000	42	14	23	8
11025	42	14	23	8
8000	39	14	19	7

4.6.9.5 TNS in the scalable coder

For the first mono layer, and for both channels of the first stereo layer a tns_data_present bit is available which activates the usage of TNS for a certain channel. The TNS filter information, however, is not necessarily transmitted along with the enable bit. Table 4.139 lists the source channel from which the TNS filter information for a specific output channel must be taken.

In all TwinVQ-mono/AAC-mono and TwinVQ-stereo/AAC-stereo configurations either the first TwinVQ layer uses TNS (**tns_data_present** == 1 in tvq_main_header(), no **tns_data_present** bit in ASME), **or** the ASME (tns_data_present == 0 in tvq_main_header(), **tns_data_present** bit in ASME), but not both at the same time..

In all TwinVQ-mono/AAC-stereo configurations the first TwinVQ layer carries the **tns_data_present** bit **and** two tns_data_present bits occure in the first AAC stereo layer (one for each audio output channel).

In all TwinVQ-mono/AAC-mono/AAC-stereo configurations either the first TwinVQ layer uses TNS and therefore the tns_data_present bit and tns_data() occures in the tvq_main_header(), **or** the first AAC mono-layer **and** in both cases two tns_data_present bits in the first AAC stereo layer.

Table 4.139 – TNS filter information source channel information depending on tns_data_present in the scalable mono / stereo coder

tns_data_present M-Channel	tns_data_present L-Channel	tns_data_present R-Channel	TNS Info M Source Chan.	TNS Info L Source Chan.	TNS Info R Source Chan.
0	0	0	-	-	-
1	0	0	M	M	M
0	1	1	-	L	R
0	1	0	-	L	-
0	0	1	-	-	R
1	0	1	M	M	R/M
1	1	0	M	L/M	M
1	1	1	M	L/M	R/M

The L/M and R/M TNS entries describe a serial layout of two TNS filters (See also the block diagrams in subclause 4.5.2.2). Additionally, the following rules apply in this case:

- The execution of the M-Filter must stop at the scale factor band, which is denoted by the max_sfb parameter of the highest mono-layer.
- The M-filter is not calculated if the lower boundary of the L-, or R-Filter, respectively, is lower than max_sfb of the highest mono layer. This allows to override the M-Filter for the lower frequency bands, if desired.

With this definition each spectral line is still filtered once at most. Therefore, the overall complexity of this combined filter is the same as the complexity of the usual single filter

If TNS is used in a scalable coder with a core coder, the TNS encoder filter of the fist mono AAC layer have to be applied to the output of the MDCT which is employed to generate the spectrum of the core coder. These encoder filters use the LPC coefficients already decoded for the corresponding TNS decoder filters. The filters are slid across the specified target frequency range exactly the way described for the decoder filter. The difference between decoder and encoder filtering is that each all-pole (auto-regressive) decoder filter used for TNS decoding is replaced by its inverse (all-zero, moving average) filter.

If TNS is used in a scalable coder with a TwinVQ coder, and if the TwinVQ layer does not use TNS, the TNS encoder filter of the fist mono AAC layer have to be applied to the output of the TwinVQ decoder. These encoder filters use the LPC coefficients already decoded for the corresponding TNS decoder filters. The filters are slid across the specified target frequency range exactly the way described for the decoder filter. The difference between decoder and encoder filtering is that each all-pole (auto-regressive) decoder filter used for TNS decoding is replaced by its inverse (all-zero, moving average) filter.

The filter equation is :

$$y[n] = x[n] + lpc[1] * x[n-1] + ... + lpc[order] * x[n-order]$$

The number of filters, filtering direction etc. is controlled exactly like in the decoding process.

4.6.9.5.1 Mono base + Stereo AAC without any AAC mono layer

If TNS is used in a scalable coder with a core coder, the TNS encoder filter of either the left or the right channel (depending on the **tns_channel_mono_layer** flag) of the first AAC stereo layer has to be applied to the output of the core coder.

If TNS is used in a scalable coder with a TwinVQ coder, and if the TwinVQ layer does not use TNS, the TNS encoder filter of either the left or the right channel (depending on the **tns_channel_mono_layer** flag) of the first AAC stereo layer have to be applied to the output of the TwinVQ decoder.

4.6.10 Spectrum normalization

4.6.10.1 Tool description

In the TwinVQ decoder, spectral de-normalization is used in combination with inverse vector quantization of the MDCT coefficients, whose reproduction has globally flat shape. Using this tool, the spectral envelope is regenerated by decoding gain, Bark-scale envelope and an envelope specified with LPC parameters. Bark-scale envelope is reconstructed using a vector quantization decoder. LPC coefficients are quantized in LSP domain by means of 2-stage split vector quantization with moving average interframe prediction. Decoded LSP coefficients are directly used for generating an amplitude spectrum (square root of the power spectral envelope).

In a long MDCT block size mode, periodic peak components are optionally added to the flattened MDCT coefficients for the low rate coder.

4.6.10.2 Definitions

a	MA prediction coefficients used for LSP quantization
alfq[][]	Predictive coefficients for Bark-envelope
AMP_MAX	maximum value of mu-law quantizer for global gain
AMP_NM	normalization factor for global gain
BASF_STEP	step size of base frequency quantizer for periodic peak components coding
bfreq[]	base frequency of periodic peak components
blim_h[]	bandwidth control factor (higher part)
blim_l[]	bandwidth control factor (lower part)
BLIM_STEP_H	number of steps of bandwidth control quantization (higher part)
BLIM_STEP_L	number of steps of bandwidth control quantization (lower part)
CUT_M_H	minimum bandwidth ratio (higher part)
CUT_M_L	maximum bandwidth ratio (lower part)
cv_env[][]	code vectors of envelope codebook
env[][][]	Bark-scale envelope projected onto Bark-scale frequency axis
fb_shift[][]	Syntax element indicating theactive frequency band of the adaptive bandwidth control.
FW_ALF_STEP	MA prediction coefficient for quantization of Bark-scale envelope
FW_CB_LEN	length of code vector of Bark-scale envelop codebook
FW_N_DIV	number of interleave division of Bark-scale envelope vector quantization
gain_p[][]	gain factors of the periodic peak components
gain[][]	gain factors of MDCT coefficients
global_gain[]	global gain of MDCT coefficients normalized by AMP_NM
index_blim_h[]	syntax element indicating higher part bandwidth control
index_blim_l[]	syntax element indicating lower part of bandwidth control
index_env[][][]	syntax elements indicating Bark-scale envelope elements
index_fw_alf[]	syntax element indicating the MA prediction switch of Bark-scale envelope quantization
index_gain[][]	syntax elements indicating global gain of MDCT coefficients

index_gain_sb[][][]	syntax elements indicating subblock gain of MDCT coefficients
index_lsp0[][]	syntax elements indicating MA prediction coefficients used for LSP quantization
index_lsp1[]	syntax element indicating the first-stage LSP quantization
index_lsp2[][]	syntax element indicating the second-stage LSP quantization
index_pgain[]	syntax elements indicating gain of periodic peak components
index_pit[][]	syntax elements indicating base frequency of periodic peak components
index_shape0_p[]	syntax element indicating peak elements quantization index for shape vector of conjugate channel 0
index_shape1_p[]	periodic peak elements quantization index for shape vector of conjugate channel 1
isp[]	split point table for second-stage LSP quantization
lengthp[]	lengths of code vectors for periodic peak components quantization
lnenv[][]	Bark-scale envelope projected onto linear-scale frequency axis
LOWER_BOUNDARY[][]	lower boundary of active frequency band used in scalable layers
lpenv[][]	LPC spectral envelope
lsp[][]	LPC coefficients which range is set from zero to π.
lyr	indicates enhancement layer number. Number 0 is assigned for the base layer.
LSP_SPLIT	number of splits of 2nd-stage vector quantization for LSP coding
MU	mu factor for mu-law quantization for gain
N_CRB	number of subbands for Bark-scale envelope coding
N_DIV_P	number of interleave division for periodic peak components coding
N_FR	number of samples in a subframe
N_FR_P	number of elements of periodic peak components
N_SF	number of subframe in a frame. The value is set in subclause 4.5.2.5.2.
p_cv_env[][]	Bark-scale envelope vector reconstructed in the previous frame
pit[]	periodic peak components
pit_seq[][][]	periodic peak components projected into linear scale
PIT_CB_SIZE	size of codebook for periodic peak components quantization
PGAIN_MAX	maximum value of mu-law quantizer for gain of periodic peak components
PGAIN_MU	mu factor for mu-law quantization for periodic peak components
PGAIN_STEP	step size of mu-law quantizer for gain of periodic peak components
pol0_p	polarity of conjugate channel 0 for periodic peak components quantization
pol1_p	polarity of conjugate channel 1 for periodic peak components quantization
pit_cv0[]	econstructed shape of conjugate channel 0 for periodic peak components quantization (Elements are given in the first 64 rows of Table 4.A.28.)
pit_cv1[]	reconstructed shape of conjugate channel 1 for periodic peak components quantization (Elements are given in the last 64 rows of Table 4.A.28)
STEP	step size of mu-law quantizer for global gain
SUB_AMP_MAX	maximum amplitude of mu-law quantizer for subframe gain ratio
SUB_AMP_NM	normalization factor for subframe gain ratio

SUB_STEP	step size of mu-law quantizer for subframe gain
subg_ratio[]	subframe gain ratio
UPPER_BOUNDARY[][]	upper boundary of active frequency band used in scalable layers
v[]	LSP coefficients
*v*1[]	reconstructed vector from 1st-stage VQ for LSP coding
*v*2[]	reconstructed vector from 2nd-stage VQ for LSP coding
x_flat[]	normalized MDCT coefficients (input)
spec[][][]	de-normalized MDCT coefficients (output)

4.6.10.3 Decoding process

The decoding process consists of five parts: gain decoding, Bark-scale envelope decoding, periodic peak components decoding, LPC spectrum decoding, and inverse normalization (see Figure 4.31).

4.6.10.3.1 Initializations

Before starting any process, prediction memories p_cv_env[][] and $\alpha_i^{(j)}$ are cleared.

4.6.10.3.2 Gain decoding

In the first step of gain decoding, the global gain is decoded using mu-law inverse quantizer described as follows:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   g_temp = index_gain * STEP + STEP / 2;
   global_gain =
      (AMP_MAX * (exp10(g_temp * log10(1.+MU) / AMP_MAX) -1) / MU) / AMP_NM;
}
```

Next, subband gain ratios are decoded using mu-law inverse quantizer described as follows:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   if (N_SF > 1) {
      for (isf = 0; isf < N_SF; isf++) {
         g_temp = index_gain_sb[i_ch][isf+1] * SUB_STEP +SUB_STEP/2.;
         subg_ratio[isf] =
           (SUB_AMP_MAX*(exp10(g_temp*log10(1.+MU)/SUB_AMP_MAX)-1)/MU)
            / SUB_AMP_NM;
      }
   }
   else {
      subg_ratio[i_ch][0] = 1;
   }
}
```

Finally, gain factors are reconstructed as follows:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   for (isf = 0; isf < N_SF; isf++) {
      gain[i_ch][isf] = global_gain[i_ch] * subg_ratio[i_ch][isf] / SUB_AMP_NM;
   }
}
```

4.6.10.3.3 Decoding of periodic peak components

Periodic peak components are optionally added to the input coefficients. The periodic peak components are coded using vector quantization. This process is active when the parameter **ppc_present** is set to TRUE. Otherwise, all the elements of output array, pit_seq[] are set to zero and the process is skipped.

```
MAXBIT_P = 6
PIT_CB_SIZE = (1 << MAXBIT_P)
```

4.6.10.3.3.1 Decoding of polarity

```
for (idiv = 0; idiv < N_DIV_P; idiv++) {
   pol0[idiv] = 2*(index_shape0_p[idiv] / PIT_CB_SIZE) - 1;
```

```
    pol1[idiv] = 2*(index_shape1_p[idiv] / PIT_CB_SIZE) - 1;
}
```

4.6.10.3.3.2 Decoding of shape code

```
for (idiv = 0; idiv < N_DIV_P; idiv++) {
    index0[idiv] = index_shape0_p[idiv] % PIT_CB_SIZE;
    index1[idiv] = index_shape1_p[idiv] % PIT_CB_SIZE;
}
```

Decoding gain of periodic peak components

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
    temp = index_pgain[i_ch] * PGAIN_STEP + PGAIN_STEP / 2;
    gain_p[i_ch] =
        (PGAIN_MAX*(exp10(temp*log10(1.+PGAIN_MU)/PGAIN_MAX)-1)/PGAIN_MU)/AMP_NM;
}
```

4.6.10.3.3.3 Reconstruction of periodic peak components

There are two steps of procedures. First the lengths of code vectors for periodic peak components, lengthp[], are calculated. Then, the periodic peak components pit[] are calculated.

```
for (idiv = 0; idiv < N_DIV_P; idiv++) {
    lengthp[idiv] = (N_FR_P*N_CH+N_DIV_P-1-idiv) / N_DIV_P;
}

for (idiv = 0; idiv < N_DIV_P; idiv++) {
    if (N_CH == 1) {
        for (icv = 0; icv < lengthp[idiv]; icv++) {
            ismp = idiv + icv * N_DIV_P;
            pit[ismp] =  (pol0[idiv]*pit_cv0[index0[idiv][icv]] +
pol1[idiv]*pit_cv1[index1[idiv]][icv]) / 2;
        }
    }
    else {
        for (icv = 0; icv < lengthp[idiv]-1; icv++) {
            ismp = ((icv+idiv)%N_DIV_P)+ icv * N_DIV_P;
            ismp = ismp/2+(ismp%2)*20;
            pit[ismp] =  (pol0[idiv]*pit_cv0[index0[idiv][icv]] +
pol1[idiv]*pit_cv1[index1[idiv]][icv]) / 2;
        }
            icv = lengthp[idiv]-1;
            ismp = idiv + icv* N_DIV_P;
            ismp = ismp/2+(ismp%2)*20;
            pit[ismp] =  (pol0[idiv]*pit_cv0[index0[idiv][icv]] +
pol1[idiv]*pit_cv1[index1[idiv]][icv]) / 2;
    }
}
```

4.6.10.3.3.4 Projecting periodic peak components into linear scale

First, parameters are calculated as following:

```
fcmin = log2((N_FR/(double)ISAMPF)*0.2);
fcmax = log2((N_FR/(double)ISAMPF)*2.4);
if (ISAMPF == 8) bandwidth = 1.5;
    else if (ISAMPF >= 11) bandwidth = 2.0;
    else if (ISAMPF >= 22) bandwidth = 4.0;
if (bandwidth < 1./UPPER_BOUNDARY[0][0] ) bandwidth = 1./UPPER_BOUNDARY[0][0];
```

where fcmin is the minimum frequency to be quantization expressed in log scale, fcmax is the maximum and ISAMPF is an integer sampling frequency in [kHz] truncated from the standard frequency values listed in the right column of Table 4.68.

Next, the base frequency of the periodic peak components is decoded according to the following procedure:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
    pow_i = (int)(pow( 1.009792f, (float)index_pit[i_sup] ) *4096.+0.5);
    bl_i = (int)((float)block_size_samples/(float)isampf * 0.2*1024+0.5);
```

```
    pitch_i = pow_i *bl_i /256.;
    bfreq[I_ch] = pitch_i/16384.;
}
```

Before projecting the perodic peak components into linear scale, all the elements of target array pit_seq[][] is set to zero:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
    for (ismp = 0; ismp < N_FR; ismp++) {
        pit_seq[i_ch][ismp] = 0.;
    }
}
```

Then, reconstructed pereodic peak components are projected to linear-scale as follows:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
    if (bandwidth * upperlimit_i < 16384) {
        tmpnp0_i = pitch_i*16384. / (upperlimit_i);
        tmpnp1_i = tmpnp0_i *N_FR_P;
        tmpnp0_i = tmpnp1_I / N_FR;
        npcount = tmpnp0_i/16384.;
      } else {
        tmpnp0_i = pitch_i * bandwidth*2;
        tmpnp1_i = tmpnp0_i *N_FR_P;
        tmpnp0_i = tmpnp1_i/N_FR;
        npcount = tmpnp0_i / 32768;
      }

    iscount=0;
    for (jj = 0; jj < npcount/2; jj++) {
            pit_seq[i_ch][jj] = pit[jj+i_ch*N_FR_P];
            iscount ++;
    }
    for (ii = 0; ii < (ntt_N_FR_P)&& (iscount<ntt_N_FR_P); ii++) {
           tmpnp0_i = pitch_i * (ii+1);
           tmpnp0_i += 8192;
           i_smp = tmpnp0_i / 16384;

        for (jj =- npcount/2; jj < (npcount-1)/2+1; jj++) {
            pit_seq[i_ch][i_smp+jj] = pit[iscount+i_ch*N_FR_P];
            iscount ++;
            if (iscount >= N_FR_P) break;
        }
    }
}
```

In case of skipping the periodic peak components decoding process, all the elements of pitch component array pit_seq[][] are set to zero.

4.6.10.3.4 Decoding of bark-scale envelope

The Bark-scale envelope is decoded in each subframe. There are two procedure stages: inverse quantizing of the envelope vectors env[][] and projecting the bark-scale envelopes env[][] onto the linear scale envelopes lnenv[][].

4.6.10.3.4.1 Inverse quantization of envelope vector

The inverse quantization part is illustrated in Figure 4.28.

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
    for (isf = 0; isf < N_SF; isf++) {
        alfq[i_ch][isf] = index_fw_alf[i_ch][isf] * FW_ALF_STEP;
        for (ifdiv = 0; ifdiv < FW_N_DIV; ifdiv++) {
            for (icv = 0; icv < FW_CB_LEN; icv++) {
                ienv = FW_N_DIV * icv + ifdiv;
                dtmp = cv_env[index_env[ich][isf][ifdiv]][icv];
                env[i_ch][isf][ienv] = dtmp + alfq[i_ch][isf] * p_cv_env[i_ch][icv] + 1;
```

```
            p_cv_env[i_ch][icv] = dtmp;
        }
      }
   }
}
```

The cv_env[][] is the Bark-scale envelope codebook listed in Table 4.A.27.

4.6.10.3.4.2 Projecting the Bark-scale envelope onto a linear scale

The envelopes env[][][] are expressed using the Bark scale on the frequency axis. The de-normalization procedure requires a linear-scale envelopes.

Before the projecting process, the boundary table of the bark-scale subband, crb_tbl[], is determined. In case of the base layer (lyr = 0),

```
if (sampling_rate <= 16000)
    use the scalefactor band table of AAC for 16 kHz up to 41st value (Table 4.115)
else
    use the scalefactor band table of AAC for 24 kHz up to 41st value (Table 4.117)
```

Number of scalefactor bands is 42 for long frame size.

42nd value is the frame length of the long frame (1024 or 960).

Barkscale estimation is not used for short frames.

If lyr >= 1, the values of the Bark-scale subband table are calculated as follows.

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   lower_band_i = (int)(AC_BTM[lyr][i_ch][fb_shift]*16384.);
   upper_band_i = (int)( AC_TOP[lyr][i_ch][fb_shift]*16384.);
   average_number_of_lines=
      (int)(frame_length*(upper_band_i-lower_band_i))/N_CRB;
       for (i = 0; i < N_CRB-1; i++) {
                  crb_tbl[i_ch][i]  =  (int)((i+1)*(i+1)* average_number_of_lines/
N_CRB/2.0)
         crb_tbl[i_ch][I]  +=  (i+1)* average_number_of_lines/2.0 +8192;
         crb_tbl[i_ch][I]  /= 16384.
                crb_tbl{i_ch][i]  += (int)(frame_length*lower_band_i)/16384.;
      }
         crb_tbl[i_ch][N_CRB-1] = (int)(frame_length*lower_band_i)/16384.
         +(int)(frame_length*(upper_band_i-lower_band_i))/16384.;
}
```

After the crb_tbl[][] is determined, the projecting process is done as follows:

```
for (i_ch = 0; i_ch < N_CH ; i_ch++){
   for (isf = 0; isf < N_SF; isf++){
      ismp = 0
      for (ienv = 0; ienv < N_CRB; ienv++) {
         while (ismp < crb_tbl[i_ch][ienv]) {
            lnenv[i_ch][isf][ismp] = env[i_ch][isf][ienv];
            ismp++;
         }
      }
   }
}
```

The values of the UPPER_BOUNDARY[][] and LOWER_BOUNDARY[][] are defined in subclause 4.6.5.4.4.

If **postprocess_present** is active, lnenv[i_ch][isf][ismp] is modified as follows:

```
   If (lnenv[i_ch][isf][ismp] < 1.0) lnenv[i_ch][isf][ismp]  = lnenv[i_ch][isf][ismp] *
lnenv[i_ch][isf][ismp];
```

4.6.10.3.5 Decoding of LPC spectrum

The LPC spectrum is represented by LSP coefficients. In the decoding process the LSP coefficients are reconstructed first; then they are transformed into the LPC spectrum, which represents the square root of the power spectrum.

4.6.10.3.5.1 LSP coefficients decoding using the MA prediction

MA prediction coefficients are determined by referring to the coefficient table $\alpha\, t_i^{(j)}$. The rule is:

$$\alpha_i^{(j)}[i\_ch] = \alpha\, t_i^{(j)}[i\_ch](index\_lsp0[i\_ch]) \qquad \text{for } i = 1 \text{ to } N\_PR,\ j = 1 \text{ to } MA\_NP,\ i\_ch = 0 \text{ to } N\_CH - 1,$$

where i is LPC order and j is MA prediction order. The coefficient table $\alpha\, t_i^{(j)}$ is listed in subclause Table 4.A.4, Table 4.A.17 and Table 4.A.18 (81th and 82th row).

4.6.10.3.5.2 First-stage inverse quantization of LSP decoding

$$v1_i[i\_ch] = lspcode1_i(index\_lsp1[i\_ch]) \qquad \text{for } i = 1 \text{ to } N\_PR,\ i\_ch = 0 \text{ to } N\_CH - 1,$$

where lspcode1 is the first-stage LSP codebook listed in subclause 4.A.4, Table 4.A.17 and Table 4.A.18 (from 1st to 64th row).

4.6.10.3.5.3 Second-stage inverse quantization of LSP decoding

$$v2_i[i\_ch] = lspcode2_i(index\_lsp2[i\_ch][k])$$
$$\text{for } k = 0 \text{ to } LSP\_SPLIT - 1,\ \ i = isp(k) + 1 \text{ to } isp(k+1) - 1,\ \ i\_ch = 0 \text{ to } N\_CH - 1\ ,$$

where lspcode2 is the second-stage LSP codebook listed in subclause 4.A.4 (from 65th to 80th row). Values of isp(k) are listed in Table 4.142.

4.6.10.3.5.4 Reconstruction of LSP coefficients

LSP coefficients lsp[][] are calculated as follows:

$$\vec{v}[i\_ch] = \vec{v1}[i\_ch] + \vec{v2}[i\_ch], \quad \text{for } i\_ch = 0 \text{ to } N\_CH - 1$$

$$\alpha_i^{(0)}[i\_ch] = 1 - \sum_{j=1}^{MA\_NP} \alpha_i^{(j)}[i\_ch] \qquad \text{for } i = 1 \text{ to } N\_PR,\ i\_ch = 0 \text{ to } N\_CH - 1$$

$$lsp[i\_ch][i] = \sum_{j=0}^{MA\_NP} \alpha_i^{(j)}[i\_ch] \cdot v_i^{(-j)}[i\_ch] \quad \text{for } i = 1 \text{ to } N\_PR,\ i\_ch = 0 \text{ to } N\_CH - 1$$

$$\vec{v}^{(j-1)}[i\_ch] = \vec{v}^{(j)}[i\_ch] \qquad \text{for } j = -MA\_NP - 1 \text{ to } 0,\ i\_ch = 0 \text{ to } N\_CH - 1$$

4.6.10.3.5.5 Transforming LSP parameters into LPC spectrum

The LPC amplitude spectrum envelope corresponding to the ii-th MDCT coefficient, lpenv[][ii], is defined as follows:

Note that lpenv[][] seems to represent the power spectrum envelope, but actually represents the amplitude envelope, since the original LPC spectral envelope is derived from the square root of the power spectrum at the encoder.

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
     for (ii = 1; ii <= N_FR-1; ii++) {
          for (i = 2, P[i_ch] = 1.0; i <= N_PR; i += 2)
               P[i_ch] *= (cos(PI*ii/N_FR)-cos(lsp[i]))^2;
          for (i = 1, Q[i_ch] = 1.0; i <= N_PR; i += 2)
              Q[i_ch] *= (cos(PI*ii/N_FR)-cos(lsp[i]))^2;
          lpenv[i_ch][ii] = 1/((1-cos(PI*ii/N_FR))*P[i_ch] + (1+cos(PI*ii/N_FR))*Q[i_ch]);
     }
}
```

In the case of long frames (N_FR == 1024, or 960), lpenv[][ii] shall be calculated only at frequency points ii that satisfy the following conditions:

```
(
        ((ii =  0 <= ii < N_FR/2) &&  (ii%4 == 0))
     || ((ii =  0 <= ii < N_FR/2-4) &&  ((ii%2 == 0) && (lpenv[][ii/4*4] > lpenv[][ii/4*4+8] )
&&
                                          (lpenv[][ii/4*4+4]*1.95 > lpenv[][ii/4*4] +
lpenv[][ii/4*4+8] )))
     || ((ii = N_FR/2 <= ii < N_FR) && (ii%8 == 0))
     || ((ii = N_FR/2 <= ii <N_FR-8) &&  ((ii%4 == 0) && (lpenv[][ii/8*8] > lpenv[][ii/8*8+16]
) &&
                                          (lpenv[][ii/8*8+8]*1.95 > lpenv[][ii/8*8] +
lpenv[][ii/8*8+16] )))
).
```

For the remaining frequency points, the lpenv[ii] values are calculated by linear interpolation from the values already calculated at the nearest two frequency points. If frequency points are larger than N_FR-8, lpenv[][ii] shall be equal to lpenv[][N_FR-8].

If this tool is used as an element of scalable coder, LPC spectrum is squeezed into the active frequency band:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   nfr_lu = UPPER_BOUNDARY[lyr][i_ch] - LOWER_BOUNDARY[lyr][i_ch];
   for (ismp = 0; ismp < LOWER_BOUNDARY[lyr][i_ch]; ismp++) {
      lpenv_tmp[i_ch][ismp] = 0;
   }
   upperband_i = (int)( AC_TOP[lyr][i_ch][fb_shift] * 16384.);
   lowerband_i = (int)( AC_BTM[lyr][i_ch][fb_shift] * 16384.);
   ftmp = (16384*16384)/( upperband_i  - lowerband_i );
   for (ismp = 0; ismp < nfr_lu; ismp++) {
      ftmp = (int)(ismp*ftmp)/16384;
      lpenv_tmp[i_ch][ismp+LOWER_BOUNDARY[lyr][i_ch]] = lpenv[i_ch][ftmp];
   }


   for (ismp = UPPER_BOUNDARY[lyr][i_ch]; ismp < N_FR; ismp++) {
      lpenv_tmp[i_ch][ismp] = 0;
   }
   for (ismp = 0; ismp < N_FR; ismp++) {
      lpenv[i_ch][ismp] = lpenv_tmp[i_ch][ismp];
   }
}
```

The values of UPPER_BOUNDARY, LOWER_BOUNDARY, AC_TOPand AC_BTM are defined in subclause 4.6.5.4.4.

4.6.10.3.6 Inverse normalization

The input coefficients x_flat[] are applied to inverse normalization according to the following procedure, and output coefficients spec[][][] are created.

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   for (isf = 0; isf < N_SF; isf++) {
      for (ismp = 0; ismp < N_FR; ismp++) {
         spec[isf][ismp] =
          (x_flat[ismp + (isf+i_ch*N_SF)*N_FR]
         *lnenv[i_ch][isf][ismp]* gain[i_ch][isf] + p_gain[i_ch]*pit_seq[i_ch][ismp])
         *lpenv[i_ch][ismp];
      }
   }
}
```

4.6.10.3.7 Bandwidth control

This functionaliry is valid only if **bandlimit_present** flag is ON.

After the inverse normalization the upper and lower bands of the output coefficients spec[][][] are set to zero.

In the bandwidth decoding modules the higher signal bandwidth ratio *blim_h*[] is decoded as follows:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   blim_h[i_ch] =
      (1. - (1.-CUT_M_H) * (double)index_blim_h[i_ch]/(double)BLIM_STEP_H))
      * UPPER_BOUNDARY[0][i_ch];
}
```

The blim_l[] is decoded as follows:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   if (index_blim_l[i_ch] == 1)
      blim_l[i_ch] = LOWER_BOUNDARY[0][i_ch]+CUT_M_L;
   else
            blim_l[i_ch] = 0;
}
```

In the bandwidth limitation module, higher and lower parts of MDCT coefficients are set to zero as follows:

```
for (i_ch = 0; i_ch < N_CH; i_ch++) {
   NbaseH = blim_h[i_ch] * N_FR;
   NbaseL = blim_l[i_ch] * N_FR;
   for (isf = 0; isf < N_SF; isf++) {
      for (ismp = NbaseH; ismp < N_FR; ismp++) {
         spec[i_ch][isf][ismp] = 0;
      }
      for (ismp = 0; ismp < NbaseL; ismp++) {
         spec[i_ch][isf][ismp] = 0;
      }
   }
}
```

4.6.10.4 Diagrams

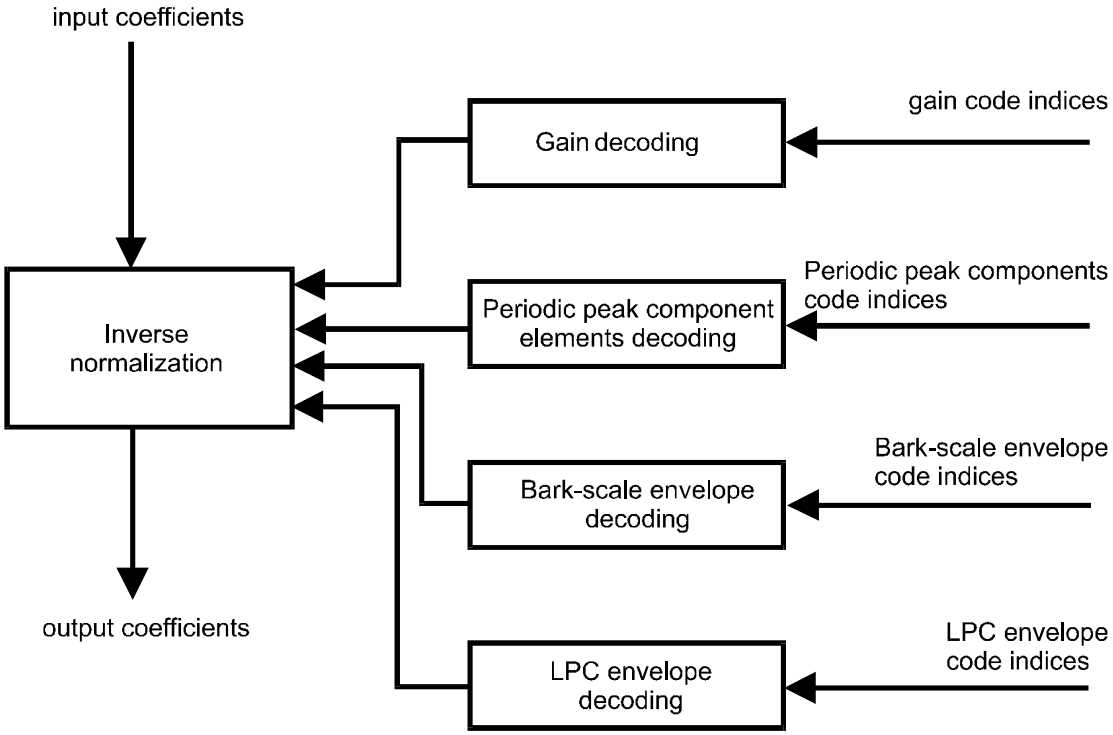


Figure 4.31 – Decoding process of the TwinVQ

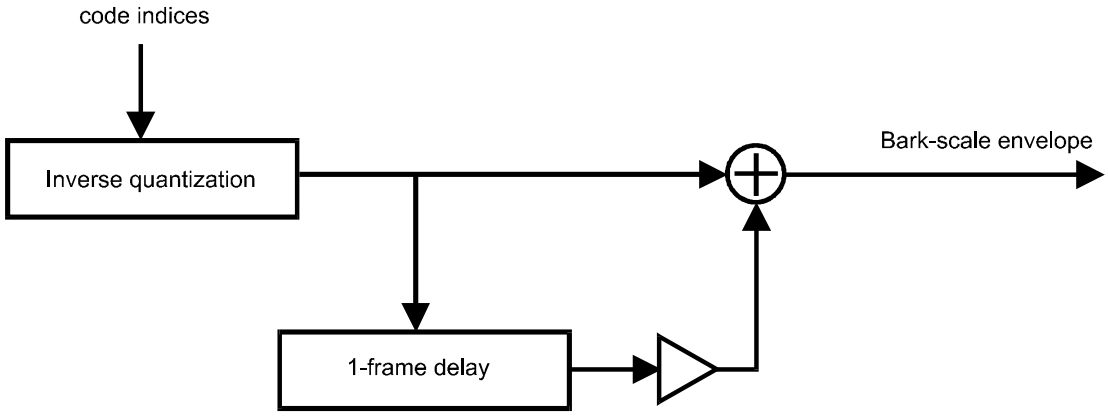


Figure 4.32 – Reconstruction process of the Bark-scale envelope

4.6.10.5 Tables

Parameters used in the inverse spectrum normalization process are set as follows. Other parameters related to the syntax are defined in subclause 4.5.2.5.3.

Table 4.140 – Parameter table for the core and core_960 mode

Parameter	Value	Meaning
Common:		
AMP_MAX	16000	Maximum amplitude in the mu-law coding of the frame gain
AMP_NM	1024	Normalized amplitude of flattened coefficients
BLIM_BITS_H	2	Bits for higher bandwidth control
BLIM_BITS_L	1	Bits for lower bandwidth control
BLIM_STEP_H	4	Number of steps of band limitation quantization
CUT_M_H	0.7	Minimum bandwidth ratio (higher part)
CUT_M_L	0.0025	Maximum bandwidth ratio (lower part)
FW_ALF_STEP	0.5	MA prediction coefficient for the envelope coding
MA_NP	1	MA prediction order (LSP coding)
MU	100	Parameter mu in the mu-law coding of the frame power
N_PR	20	LPC order
NUM_STEP	512	Number of gain-quantization steps
STEP	AMP_MAX / (NUM_STEP-1) = 31.31	Step width of gain-quantization
SUB_AMP_MAX	4700	Maximum of subframe-gain to frame-gain ratio
SUB_AMP_NM	1024	Normalized subframe amplitude of flattened coefficients
SUB_GAIN_BIT	4	Number of bits for the subframe gain coding
SUB_NUM_STEP	16	Number of sub-gain-quantization steps
SUB_STEP	SUB_AMP_MAX/ (SUB_NUM_STEP-1)= 313.3	Step width of sub-gain-quantization
PGAIN_MAX	20000	maximum value of mu-law quantizer for gain of ppc
PGAIN_MU	200	mu factor for mu-law quantization for ppc
PGAIN_STEP	PGAIN_MAX /(1<<PGAIN_BIT)	step size of mu-law quantizer for gain of ppc
Long block:		
FW_CB_LEN	6	Envelope code vector length
FW_N_BIT	6	Envelope code bits
FW_N_DIV	7	Number of divisions in the envelope coding
N_CRB	42	Number of bark-scale subbands
N_FR	1024/960	MDCT block size

Short block:		
N_FR	128/120	MDCT block size

Table 4.141 – Parameter table for the enhance and enhance_960 mode

Parameter	Value	Meaning
Common:		
AMP_MAX	8000	Maximum amplitude in the mu-law coding of the frame gain
AMP_NM	1024	Normalized amplitude of flattened coefficients
FW_ALF_STEP	0.5	MA prediction coefficient for the envelope coding
MA_NP	1	MA prediction order (LSP coding)
MU	100	Parameter mu in the mu-law coding of the frame power
N_PR	20	LPC order
NUM_STEP	256	Number of gain-quantization steps
STEP	AMP_MAX / (NUM_STEP-1) = 31.37	Step width of gain-quantization
SUB_AMP_MAX	6000	Maximum of subframe-gain to frame-gain ratio
SUB_AMP_NM	1024	Normalized subframe amplitude of flattened coefficients
SUB_GAIN_BIT	4	Number of bits for the subframe gain coding
SUB_NUM_STEP	16	Number of sub-gain-quantization steps
SUB_STEP	SUB_AMP_MAX/ (SUB_NUM_STEP-1)= 400.0	Step width of sub-gain-quantization
Long block:		
FW_CB_LEN	7	Envelope code vector length
FW_N_BIT	6	Envelope code bits
FW_N_DIV	7	Number of division in the envelope coding
N_CRB	42	Number of bark-scale subbands
N_FR	1024/960	MDCT block size
Short block:		
N_FR	128/120	MDCT block size

Table 4.142 – Values of isp[]

split_num	isp[split_num]
0	0
1	5
2	14
3	20

4.6.11 Filterbank and block switching

(Similar to ISO/IEC 13818-7)

4.6.11.1 Tool description

The time/frequency representation of the signal is mapped onto the time domain by feeding it into the filterbank module. This module consists of an inverse modified discrete cosine transform (IMDCT), and a window and an overlap-add function. In order to adapt the time/frequency resolution of the filterbank to the characteristics of the input signal, a block switching tool is also adopted. N represents the window length, where N is a function of the **window_sequence** (see subclauses 4.5.2.3.3). For each channel, the N/2 time-frequency values $X_{i,k}$ are transformed into the N time domain values $x_{i,n}$ via the IMDCT. After applying the window function, for each channel, the first half of the $z_{i,n}$ sequence is added to the second half of the previous block windowed sequence $z_{(i-1),n}$ to reconstruct the output samples for each channel $out_{i,n}$.

4.6.11.2 Definitions

window_sequence 2 bit indicating which window sequence (i.e. block size) is used.

window_shape 1 bit indicating which window function is selected.

Table 4.109 shows the four **window_sequences** (ONLY_LONG_SEQUENCE, LONG_START_SEQUENCE, EIGHT_SHORT_SEQUENCE, LONG_STOP_SEQUENCE).

4.6.11.3 Decoding process

4.6.11.3.1 IMDCT

The analytical expression of the IMDCT is:

$$x_{i,n} = \frac{2}{N}\sum_{k=0}^{\frac{N}{2}-1} spec[i][k]\cos\left(\frac{2\pi}{N}\left(n+n_0\right)\left(k+\frac{1}{2}\right)\right) \quad \text{for} \quad 0 \le n < N$$

where:

n = sample index

i = window index

k = spectral coefficient index

N = window length based on the window_ sequence value

n_0 = (N / 2 + 1) / 2

The synthesis window length N for the inverse transform is a function of the syntax element **window_sequence** and the algorithmic context. It is defined as follows:

Window length 2048:

$$N = \begin{cases} 2048, & \text{if ONLY\_LONG\_SEQUENCE (0x0)} \\ 2048, & \text{if LONG\_START\_SEQUENCE (0x1)} \\ 256, & \text{if EIGHT\_SHORT\_SEQUENCE (0x2), (8 times)} \\ 2048, & \text{if LONG\_STOP\_SEQUENCE (0x3)} \end{cases}$$

Window length 1920:

$$N = \begin{cases} 1920, & \text{if ONLY\_LONG\_SEQUENCE (0x0)} \\ 1920, & \text{if LONG\_START\_SEQUENCE (0x1)} \\ 240, & \text{if EIGHT\_SHORT\_SEQUENCE (0x2), (8 times)} \\ 1920, & \text{if LONG\_STOP\_SEQUENCE (0x3)} \end{cases}$$

The meaningful block transitions are as follows:

from ONLY_LONG_SEQUENCE	to	{ONLY_LONG_SEQUENCE LONG_START_SEQUENCE
from LONG_START_SEQUENCE	to	{EIGHT_SHORT_SEQUENCE LONG_STOP_SEQUENCE
from LONG_STOP_SEQUENCE	to	{ONLY_LONG_SEQUENCE LONG_START_SEQUENCE

from EIGHT_SHORT_SEQUENCE	to	{EIGHT_SHORT_SEQUENCE LONG_STOP_SEQUENCE

In addition to the meaningful block transitions the following transitions are possible:

from ONLY_LONG_SEQUENCE	to	{EIGHT_SHORT_SEQUENCE LONG_STOP_SEQUENCE
from LONG_START_SEQUENCE	to	{ONLY_LONG_SEQUENCE LONG_START_SEQUENCE
from LONG_STOP_SEQUENCE	to	{EIGHT_SHORT_SEQUENCE LONG_STOP_SEQUENCE
from EIGHT_SHORT_SEQUENCE	to	{ONLY_LONG_SEQUENCE LONG_START_SEQUENCE

This will still result in a reasonably smooth transition from one block to the next.

4.6.11.3.2 Windowing and block switching

Depending on the **window_sequence** and **window_shape** element different transform windows are used. A combination of the window halves described as follows offers all possible window_sequences.

For **window_shape** == 1, the window coefficients are given by the Kaiser - Bessel derived (KBD) window as follows:

$$W_{KBD\_LEFT,N}(n) = \sqrt{\frac{\sum_{p=0}^{n}\left[W'(p,\alpha)\right]}{\sum_{p=0}^{N/2}\left[W'(p,\alpha)\right]}} \quad \text{for} \quad 0 \le n < \frac{N}{2}$$

$$W_{KBD\_RIGHT,N}(n) = \sqrt{\frac{\sum_{p=0}^{N-n-1}\left[W'(p,\alpha)\right]}{\sum_{p=0}^{N/2}\left[W'(p,\alpha)\right]}} \quad \text{for} \quad \frac{N}{2} \le n < N$$

where:

W', Kaiser - Bessel kernel window function, see also [5], is defined as follows:

$$W'(n,\alpha) = \frac{I_0\left[\pi\alpha\sqrt{1.0-\left(\frac{n-N/4}{N/4}\right)^2}\right]}{I_0\left[\pi\alpha\right]} \qquad \text{for } 0 \le n \le \frac{N}{2}$$

$$I_0[x] = \sum_{k=0}^{\infty} \left[\frac{\left(\frac{x}{2}\right)^k}{k!} \right]^2$$

α = kernel window alpha factor, $\alpha = \begin{cases} 4 \text{ for N} = 2048\ (1920) \\ 6 \text{ for N} = 256\ (240) \end{cases}$

Otherwise, for **window_shape ==** 0, a sine window is employed as follows:

$$W_{SIN\_LEFT,N}(n) = \sin\left(\frac{\pi}{N}(n+\frac{1}{2})\right) \qquad \text{for} \quad 0 \le n < \frac{N}{2}$$

$$W_{SIN\_RIGHT,N}(n) = \sin\left(\frac{\pi}{N}(n+\frac{1}{2})\right) \qquad \text{for} \quad \frac{N}{2} \le n < N$$

The window length N can be 2048 (1920) or 256 (240) for the KBD and the sine window. How to obtain the possible window sequences is explained in the parts a)-d) of this subclause. All four window_sequences described below have a total length of 2048 samples.

For all kinds of window_sequences the window_shape of the left half of the first transform window is determined by the window shape of the previous block. The following formula expresses this fact:

$$W_{LEFT,N}(n) = \begin{cases} W_{KBD\_LEFT,N}(n), \text{ if } \ window\_shape\_previous\_block == 1 \\ W_{SIN\_LEFT,N}(n), \text{ if } \ window\_shape\_previous\_block == 0 \end{cases}$$

where:

window_shape_previous_block: **window_shape** of the previous block (i-1).

For the first raw_data_block() to be decoded the **window_shape** of the left and right half of the window are identical.

a) ONLY_LONG_SEQUENCE:

The **window_sequence ==** ONLY_LONG_SEQUENCE is equal to one LONG_WINDOW with a total window length N_l of 2048 (1920).

For **window_shape ==** 1 the window for ONLY_LONG_SEQUENCE is given as follows:

$$W(n) = \begin{cases} W_{LEFT,N\_l}(n), & \text{for } 0 \le \mathrm{n} < N\_l/2 \\ W_{KBD\_RIGHT,N\_l}(n), & \text{for } N\_\mathrm{l}/2 \le \mathrm{n} < \mathrm{N}\_l \end{cases}$$

If **window_shape ==** 0 the window for ONLY_LONG_SEQUENCE can be described as follows:

$$W(n) = \begin{cases} W_{LEFT,N\_l}(n), & \text{for } 0 \le \mathrm{n} < N\_l/2 \\ W_{SIN\_RIGHT,N\_l}(n), & \text{for } N\_l/2 \le \mathrm{n} < N\_l \end{cases}$$

After windowing, the time domain values ($z_{i,n}$) can be expressed as:

$$z_{i,n} = w(n) \cdot x_{i,n};$$

b) LONG_START_SEQUENCE:

The LONG_START_SEQUENCE is needed to obtain a correct overlap and add for a block transition from a ONLY_LONG_SEQUENCE to a EIGHT_SHORT_SEQUENCE.

Window length N_l and N_s is set to 2048 (1920) and 256 (240) respectively.

If **window_shape** == 1 the window for LONG_START_SEQUENCE is given as follows:

$$W(n) = \begin{cases} W_{LEFT,N\_l}(n), & \text{for } 0 \le n < N\_l/2 \\ 1.0, & \text{for } N\_l/2 \le n < \frac{3N\_l-N\_s}{4} \\ W_{KBD\_RIGHT,N\_s}(n+\frac{N\_s}{2}-\frac{3N\_l-N\_s}{4}), & \text{for } \frac{3N\_l-N\_s}{4} \le n < \frac{3N\_l+N\_s}{4} \\ 0.0, & \text{for } \frac{3N\_l+N\_s}{4} \le n < N\_l \end{cases}$$

If **window_shape** == 0 the window for LONG_START_SEQUENCE looks like:

$$W(n) = \begin{cases} W_{LEFT,N\_l}(n), & \text{for } 0 \le n < N\_l/2 \\ 1.0, & \text{for } N\_l/2 \le n < \frac{3N\_l-N\_s}{4} \\ W_{SIN\_RIGHT,N\_s}(n+\frac{N\_s}{2}-\frac{3N\_l-N\_s}{4}), & \text{for } \frac{3N\_l-N\_s}{4} \le n < \frac{3N\_l+N\_s}{4} \\ 0.0, & \text{for } \frac{3N\_l+N\_s}{4} \le n < N\_l \end{cases}$$

The windowed time-domain values can be calculated with the formula explained in a).

c) EIGHT_SHORT

The **window_sequence** == EIGHT_SHORT comprises eight overlapped and added SHORT_WINDOWs with a length N_s of 256 (240) each. The total length of the window_sequence together with leading and following zeros is 2048 (1920). Each of the eight short blocks are windowed separately first. The short block number is indexed with the variable j = 0,…, M-1 (M=N_l/N_s).

The **window_shape** of the previous block influences the first of the eight short blocks ($W_0(n)$) only. If **window_shape** == 1 the window functions can be given as follows:

$$W_0(n) = \begin{cases} W_{LEFT,N\_s}(n), & \text{for } 0 \le n < N\_s/2 \\ W_{KBD\_RIGHT,N\_s}(n), & \text{for } N\_s/2 \le n < N\_s \end{cases}$$

$$W_{1-(M-1)}(n) = \begin{cases} W_{KBD\_LEFT,N\_s}(n) & \text{for } 0 \le n < N\_s/2 \\ W_{KBD\_RIGHT,N\_s}(n), & \text{for } N\_s/2 \le n < N\_s \end{cases}$$

Otherwise, if **window_shape** == 0, the window functions can be described as:

$$W_0(n) = \begin{cases} W_{LEFT,N\_s}(n), & \text{for } 0 \le n < N\_s/2 \\ W_{SIN\_RIGHT,N\_s}(n), & \text{for } N\_s/2 \le n < N\_s \end{cases}$$

$$W_{1-(M-1)}(n) = \begin{cases} W_{SIN\_LEFT,N\_s}(n), & \text{for } 0 \le n < N\_s/2 \\ W_{SIN\_RIGHT,N\_s}(n), & \text{for } N\_s/2 \le n < N\_s \end{cases}$$

The overlap and add between the EIGHT_SHORT **window_sequence** resulting in the windowed time domain values $z_{i,n}$ is described as follows:

$$z_{i,n} = \begin{cases} 0, & \text{for } 0 \le n < \frac{N\_l - N\_s}{4} \\ x_{0,n-\frac{N\_l-N\_s}{4}} \cdot W_0(n - \frac{N\_l-N\_s}{4}), & \text{for } \frac{N\_l-N\_s}{4} \le n < \frac{N\_l+N\_s}{4} \\ x_{j-1,n-\frac{N\_l+(2j-3)\cdot N\_s}{4}} \cdot W_{j-1}(n - \frac{N\_l+(2j-3)N\_s}{4}) + x_{j,n-\frac{N\_l+(2j-1)N\_s}{4}} \cdot W_j(n - \frac{N\_l+(2j-1)N\_s}{4}), & \\ & \text{for } 1 \le j < M, \ \frac{N\_l+(2j-1)N\_s}{4} \le n < \frac{N\_l+(2j+1)N\_s}{4} \\ x_{M-1,n-\frac{N\_l+(2M-3)N\_s}{4}} \cdot W_{M-1}(n - \frac{N\_l+(2M-3)N\_s}{4}), & \\ & \text{for } \frac{N\_l+(2M-1)N\_s}{4} \le n < \frac{N\_l+(2M+1)N\_s}{4} \\ 0, & \text{for } \frac{N\_l+(2M+1)N\_s}{4} \le n < N\_l \end{cases}$$

d) LONG_STOP_SEQUENCE

This window_sequence is needed to switch from a EIGHT_SHORT_SEQUENCE back to a ONLY_LONG_SEQUENCE.

If **window_shape** == 1 the window for LONG_STOP_SEQUENCE is given as follows:

$$W(n) = \begin{cases} 0.0, & \text{for } 0 \le \mathrm{n} < \frac{N\_l-N\_s}{4} \\ W_{LEFT,N\_s}(n - \frac{N\_l-N\_s}{4}), & \text{for } \frac{N\_l-N\_s}{4} \le \mathrm{n} < \frac{N\_l+N\_s}{4} \\ 1.0, & \text{for } \frac{N\_l+N\_s}{4} \le \mathrm{n} < N\_l/2 \\ W_{KBD\_RIGHT,N\_l}(n), & \text{for } N\_l/2 \le \mathrm{n} < N\_l \end{cases}$$

If **window_shape** == 0 the window for LONG_START_SEQUENCE is determined by:

$$W(n) = \begin{cases} 0.0, & \text{for } 0 \le \mathrm{n} < \frac{N\_l-N\_s}{4} \\ W_{LEFT,N\_s}(n - \frac{N\_l-N\_s}{4}), & \text{for } \frac{N\_l-N\_s}{4} \le \mathrm{n} < \frac{N\_l+N\_s}{4} \\ 1.0, & \text{for } \frac{N\_l+N\_s}{4} \le \mathrm{n} < N\_l/2 \\ W_{SIN\_RIGHT,N\_l}(n), & \text{for } N\_l/2 \le \mathrm{n} < N\_l \end{cases}$$

The windowed time domain values can be calculated with the formula explained in a).

4.6.11.3.3 Overlapping and adding with previous window sequence

Besides the overlap and add within the EIGHT_SHORT **window_sequence** the first (left) half of every **window_sequence** is overlapped and added with the second (right) half of the previous **window_sequence** resulting in the final time domain values $out_{i,n}$. The mathematic expression for this operation can be described as follows. It is valid for all four possible window_sequences.

$$out_{i,n} = z_{i,n} + z_{i-1,n+\frac{N}{2}}; \qquad \text{for } 0 \le \mathrm{n} < \frac{\mathrm{N}}{2}, \quad \mathrm{N} = 2048\ (1920)$$

4.6.12 Gain Control

4.6.12.1 Tool description

The gain control tool is made up of several gain compensators and overlap/add processing stages, and an IPQF (Inverse Polyphase Quadrature Filter) stage. This tool receives non-overlapped signal sequences provided by the IMDCT stages, window_sequence and gain_control_data, and then reproduces the output PCM data. The block diagram for the gain control tool is shown in Figure 4.33.

Due to the characteristics of the PQF filterbank, the order of the MDCT coefficients in each even PQF band must be reversed. This is done by reversing the spectral order of the MDCT coefficients, i.e. exchanging the higher frequency MDCT coefficients with the lower frequency MDCT coefficients.

If the gain control tool is used, the configuration of the filter bank tool is changed as follows. In the case of an EIGHT_SHORT_SEQUENCE window_sequence, the number of coefficients for the IMDCT is 32 instead of 128 and eight IMDCTs are carried out. In the case of other window_sequence values, the number of coefficients for the IMDCT is 256 instead of 1024 and one IMDCT is performed. In all cases, the filter bank tool outputs a total of 2048 non-overlapped values per frame. These values are supplied to the gain control tool as $U_{W,B}(j)$ defined in subclause 4.6.12.3.3.

The IPQF combines four uniform frequency bands and produces a decoded time domain output signal. The aliasing components introduced by the PQF in the encoder are cancelled by the IPQF.

The gain values for each band can be controlled independently except for the lowest frequency band. The step size of gain control is 2 ^ n where n is an integer.

The gain control tool outputs a time signal sequence, which is $AS(n)$ defined in subclause 4.6.12.3.4.

4.6.12.2 Definitions

gain control data	side information indicating the gain values and the positions used for the gain change.
IPQF band	each split band of IPQF.
adjust_num	3-bit field indicating the number of gain changes for each IPQF band. The maximum number of gain changes is seven
max_band	2-bit field indicating the number of IPQF bands in which their signal gain have been controlled. The meanings of this value are shown below: 0: no bands have activated gain control. 1: signal gain on 2nd IPQF band has been controlled. 2: signal gain on 2nd and 3rd IPQF bands have been controlled. 3: signal gain on 2nd, 3rd and 4th IPQF bands have been controlled.
alevcode	4-bit field indicating the gain value for one gain change.
aloccode	2-, 4-, or 5-bit field indicating the position for one gain change. The length of this data varies depending on the window sequence.

4.6.12.3 Decoding process

The following four processes are required for decoding.

(1) Gain control data decoding

(2) Gain control function setting

(3) Gain control windowing and overlapping

(4) Synthesis filter

4.6.12.3.1 Gain control data decoding

Gain control data are reconstructed as follows.

(1)

$$NAD_{W,B} = \mathbf{adjust\_num}[B][W]$$

(2)

$$ALOC_{W,B}(m) = AdjLoc(\mathbf{aloccode}[B][W][m-1]), \quad 1 \le m \le NAD_{W,B}$$

$$ALEV_{W,B}(m) = 2^{AdjLev(\mathbf{alevcode}[B][W][m-1])}, \quad 1 \le m \le NAD_{W,B}$$

(3)

$$ALOC_{W,B}(0) = 0$$

$$ALEV_{W,B}(0) = \begin{cases} 1, & if\ NAD_{W,B} == 0 \\ ALEV_{W,B}(1), & otherwise \end{cases}$$

(4)

$$ALOC_{W,B}(NAD_{W,B}+1) = \begin{cases} 256, & W == 0 \quad if\ \text{ONLY\_LONG\_SEQUENCE} \\ \left.\begin{matrix} 112, & W == 0 \\ 32, & W == 1 \end{matrix}\right\} & if\ \text{LONG\_START\_SEQUENCE} \\ 32, & 0 \le W \le 7 \quad if\ \text{EIGHT\_SHORT\_SEQUENCE} \\ \left.\begin{matrix} 112, & W == 0 \\ 256, & W == 1 \end{matrix}\right\} & if\ \text{LONG\_STOP\_SEQUENCE} \end{cases}$$

$$ALEV_{W,B}(NAD_{W,B}+1) = 1$$

where

$NAD_{W,B}$: Gain Control Information Number, an integer

$ALOC_{W,B}(m)$: Gain Control Location, an integer

$ALEV_{W,B}(m)$: Gain Control Level, an integer-valued real number

B: Band ID, an integer from 1 to 3

W: Window ID, an integer from 0 to 7

m: an integer

aloccode[*B*][*W*][*m*] must be set so that $\{ALOC_{W,B}(m)\}$ satisfies the following conditions.

$$ALOC_{W,B}(m_1) < ALOC_{W,B}(m_2), \quad 1 \le m_1 < m_2 \le NAD_{W,B}+1$$

In cases of LONG_START_SEQUENCE and LONG_STOP_SEQUENCE, the values 14 and 15 of **aloccode**[*B*][*0*][*m*] are invalid. *AdjLoc*() is defined in Table 4.143. *AdjLev*() is defined in Table 4.144.

4.6.12.3.2 Gain control function setting

The Gain control function is obtained as follows.

(1)

$$M_{W,B,j} = Max\{m: ALOC_{W,B}(m) \le j\},$$

$$0 \le j \le 255, \quad W == 0 \quad if\ \text{ONLY\_LONG\_SEQUENCE}$$

$$\left.\begin{matrix} 0 \le j \le 111, & W == 0 \\ 0 \le j \le 31, & W == 1 \end{matrix}\right\} \quad if\ \text{LONG\_START\_SEQUENCE}$$

$$0 \le j \le 31, \quad 0 \le W \le 7 \quad if\ \text{EIGHT\_SHORT\_SEQUENCE}$$

$$\left.\begin{matrix} 0 \le j \le 111, & W == 0 \\ 0 \le j \le 255, & W == 1 \end{matrix}\right\} \quad if\ \text{LONG\_STOP\_SEQUENCE}$$

(2)

$$FMD_{W,B}(j) = \begin{cases} Inter\begin{pmatrix} ALEV_{W,B}(M_{W,B,j}), \\ ALEV_{W,B}(M_{W,B,j}+1), \\ j - ALOC_{W,B}(M_{W,B,j}) \end{pmatrix}, \\ \qquad\qquad if\ ALOC_{W,B}(M_{W,B,j}) \le j \le ALOC_{W,B}(M_{W,B,j}) + 7 \\ ALEV_{W,B}(M_{W,B,j}+1), \quad otherwise \end{cases}$$

(3)

if ONLY_LONG_SEQUENCE

$$GMF_{0,B}(j) = \begin{cases} ALEV_{0,B}(0) \times PFMD_B(j), & 0 \le j \le 255 \\ FMD_{0,B}(j-256), & 256 \le j \le 511 \end{cases}$$

$$PFMD_B(j) = FMD_{0,B}(j), \quad 0 \le j \le 255$$

if LONG_START_SEQUENCE

$$GMF_{0,B}(j) = \begin{cases} ALEV_{0,B}(0) \times ALEV_{1,B}(0) \times PFMD_B(j), & 0 \le j \le 255 \\ ALEV_{1,B}(0) \times FMD_{0,B}(j-256), & 256 \le j \le 367 \\ FMD_{1,B}(j-368), & 368 \le j \le 399 \\ 1, & 400 \le j \le 511 \end{cases}$$

$$PFMD_B(j) = FMD_{1,B}(j), \quad 0 \le j \le 31$$

if EIGHT_SHORT_SEQUENCE

$$GMF_{W,B}(j) = \begin{cases} ALEV_{W,B}(0) \times PFMD_B(j), & W == 0, \quad 0 \le j \le 31 \\ ALEV_{W,B}(0) \times FMD_{W-1,B}(j), & 1 \le W \le 7, \quad 0 \le j \le 31 \\ FMD_{W,B}(j-32), & 0 \le W \le 7, \quad 32 \le j \le 63 \end{cases}$$

$$PFMD_B(j) = FMD_{7,B}(j), \quad 0 \le j \le 31$$

if LONG_STOP_SEQUENCE

$$GMF_{0,B}(j) = \begin{cases} 1, & 0 \le j \le 111 \\ ALEV_{0,B}(0) \times ALEV_{1,B}(0) \times PFMD_B(j-112), & 112 \le j \le 143 \\ ALEV_{1,B}(0) \times FMD_{0,B}(j-144), & 144 \le j \le 255 \\ FMD_{1,B}(j-256), & 256 \le j \le 511 \end{cases}$$

$$PFMD_B(j) = FMD_{1,B}(j), \quad 0 \le j \le 255$$

(4)

$$AD_{W,B}(j) = \frac{1}{GMF_{W,B}(j)},$$

$0 \le j \le 511, \quad W == 0 \quad if$ ONLY_LONG_SEQUENCE

$0 \le j \le 511, \quad W == 0 \quad if$ LONG_START_SEQUENCE

$0 \le j \le 63, \quad 0 \le W \le 7 \quad if$ EIGHT_SHORT_SEQUENCE

$0 \le j \le 511, \quad W == 0 \quad if$ LONG_STOP_SEQUENCE

where

$FMD_{W,B}(j)$: Fragment Modification Function, a real number

$PFMD_B(j)$: Fragment Modification Function of previous frame, a real number

$GMF_{W,B}(j)$: Gain Modification Function, a real number

$AD_{W,B}(j)$: Gain Control Function, a real number

$ALOC_{W,B}(m)$: Gain Control Location defined in subclause 4.6.12.3.1, an integer

$ALEV_{W,B}(m)$: Gain Control Level defined in subclause 4.6.12.3.1, an integer-valued real number

B: Band ID, an integer from 1 to 3

W: Window ID, an integer from 0 to 7

$M_{W,B,j}$: an integer

m: an integer

and

$$Inter(a,b,j) = 2^{\frac{(8-j)\log_2(a)+j\log_2(b)}{8}}$$

Note that the initial value of $PFMD_B(j)$ must be set to 1.0.

4.6.12.3.3 Gain control windowing and overlapping

Band Sample Data are obtained through the processes (1) to (2) shown below.

(1) Gain Control Windowing

if B = 0

$$T_{W,B}(j) = U_{W,B}(j),$$

$0 \le j \le 511, \quad W == 0 \quad if$ ONLY_LONG_SEQUENCE

$0 \le j \le 511, \quad W == 0 \quad if$ LONG_START_SEQUENCE

$0 \le j \le 63, \quad 0 \le W \le 7 \quad if$ EIGHT_SHORT_SEQUENCE

$0 \le j \le 511, \quad W == 0 \quad if$ LONG_STOP_SEQUENCE

else

$$T_{W,B}(j) = AD_{W,B}(j) \times U_{W,B}(j),$$

$0 \le j \le 511, \quad W == 0 \quad if$ ONLY_LONG_SEQUENCE

$0 \le j \le 511, \quad W == 0 \quad if$ LONG_START_SEQUENCE

$0 \le j \le 63, \quad 0 \le W \le 7 \quad if$ EIGHT_SHORT_SEQUENCE

$0 \le j \le 511, \quad W == 0 \quad if$ LONG_STOP_SEQUENCE

(2) Overlapping

if ONLY_LONG_SEQUENCE

$$V_B(j) = PT_B(j) + T_{0,B}(j), \quad 0 \le j \le 255$$

$$PT_B(j) = T_{0,B}(j+256), \quad 0 \le j \le 255$$

if LONG_START_SEQUENCE

$$V_B(j) = PT_B(j) + T_{0,B}(j), \quad 0 \le j \le 255$$

$$V_B(j+256) = T_{0,B}(j+256), \quad 0 \le j \le 111$$

$$PT_B(j) = T_{0,B}(j+368), \quad 0 \le j \le 31$$

if EIGHT_SHORT_SEQUENCE

$$V_B(j) = PT_B(j) + T_{W,B}(j), \quad W == 0, \quad 0 \le j \le 31$$

$$V_B(32W+j) = T_{W-1,B}(j+32) + T_{W,B}(j), \quad 1 \le W \le 7, \quad 0 \le j \le 31$$

$$PT_B(j) = T_{w,B}(j+32), \quad W == 7, \quad 0 \le j \le 31$$

if LONG_STOP_SEQUENCE

$$V_B(j) = PT_B(j) + T_{0,B}(j+112), \quad 0 \le j \le 31$$

$$V_B(j+32) = T_{0,B}(j+144), \quad 0 \le j \le 111$$

$$PT_B(j) = T_{0,B}(j+256), \quad 0 \le j \le 255$$

where

$U_{W,B}(j)$: Band Spectrum Data, a real number

$T_{W,B}(j)$: Gain Controlled Block Sample Data, a real number

$PT_B(j)$: Gain Controlled Block Sample Data of previous frame, a real number

$V_B(j)$: Band Sample Data, a real number

$AD_{W,B}(j)$: Gain Control Function defined in subclause 4.6.12.3.2 a real number

B: Band ID, an integer from 0 to 3

W: Window ID, an integer from 0 to 7

j: an integer

Note that the initial value of $PT_B(j)$ must be set to 0.0.

4.6.12.3.4 Synthesis filter

Audio Sample Data are obtained from the following equations.

(1)

$$\tilde{V}_B(j) = \begin{cases} V_B(k), & if\ j == 4k, \\ 0, & else \end{cases} \quad 0 \le B \le 3$$

(2)

$$Q_B(j) = Q(j) \times \cos\left(\frac{(2B+1)(2j-3)\pi}{16}\right), \quad 0 \le j \le 95, \quad 0 \le B \le 3$$

(3)

$$AS(n) = \sum_{B=0}^{3} \sum_{j=0}^{95} Q_B(j) \times \tilde{V}_B(n-j)$$

where

$AS(n)$: Audio Sample Data

$V_B(n)$: Band Sample Data defined in subclause 4.6.12.3.3, a real number

$\tilde{V}_B(j)$: Interpolated Band Sample Data, a real number

$Q_B(j)$: Synthesis Filter Coefficients, a real number

$Q(j)$: Prototype Coefficients given below, a real number

B: Band ID, an integer from 0 to 3

W: Window ID, an integer from 0 to 7

n: an integer

j: an integer

k: an integer

The values of *Q*(0) to *Q*(47) are shown in Table 4.145. The values of *Q*(48) to *Q*(95) are obtained from the following equation.

$$Q(j) = Q(95 - j), \quad 48 \le j \le 95$$

4.6.12.4 Diagrams

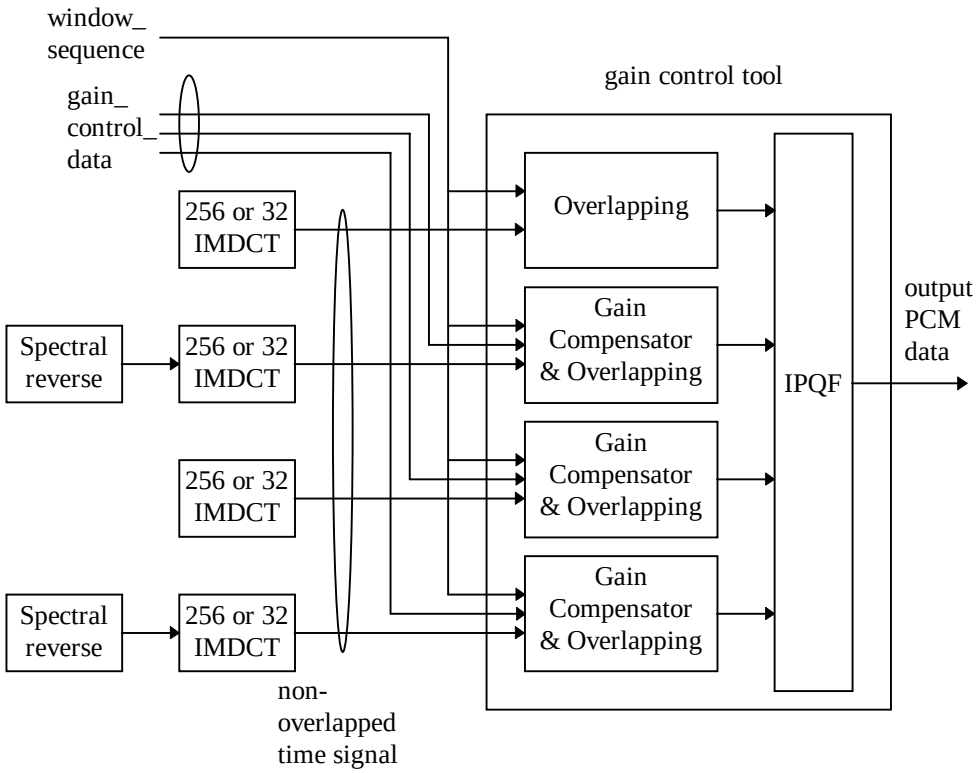


Figure 4.33 – Block diagram of gain control tool

4.6.12.5 Tables

Table 4.143 – AdjLoc()

AC	*AdjLoc(AC)*	*AC*	*AdjLoc(AC)*
0	0	16	128
1	8	17	136
2	16	18	144

3	24	19	152
4	32	20	160
5	40	21	168
6	48	22	176
7	56	23	184
8	64	24	192
9	72	25	200
10	80	26	208
11	88	27	216
12	96	28	224
13	104	29	232
14	112	30	240
15	120	31	248

Table 4.144 – AdjLev()

AV	*AdjLev(AV)*
0	-4
1	-3
2	-2
3	-1
4	0
5	1
6	2
7	3
8	4
9	5
10	6
11	7
12	8
13	9
14	10
15	11

Table 4.145 – Q()

j	*Q(j)*	*j*	*Q(j)*
0	9.7655291007575512E-05	24	-2.2656858741499447E-02
1	1.3809589379038567E-04	25	-6.8031113858963354E-03
2	9.8400749256623534E-05	26	1.5085400948280744E-02
3	-8.6671544782335723E-05	27	3.9750993388272739E-02
4	-4.6217998911921346E-04	28	6.2445363629436743E-02
5	-1.0211814095158174E-03	29	7.7622327748721326E-02
6	-1.6772149340010668E-03	30	7.9968338496132926E-02
7	-2.2533338951411081E-03	31	6.5615493068475583E-02
8	-2.4987888343213967E-03	32	3.3313658300882690E-02
9	-2.1390815966761882E-03	33	-1.4691563058190206E-02
10	-9.5595397454597772E-04	34	-7.2307890475334147E-02
11	1.1721211530118943E-03	35	-1.2993222541703875E-01
12	3.9091309127348584E-03	36	-1.7551641029040532E-01
13	6.9635703420118673E-03	37	-1.9626543957670528E-01
14	9.5595442159478339E-03	38	-1.8073330670215029E-01
15	1.0815766540021360E-02	39	-1.2097653136035738E-01
16	9.8770514991715300E-03	40	-1.4377370758549035E-02
17	6.1562567291327357E-03	41	1.3522730742860303E-01
18	-4.1793946063629710E-04	42	3.1737852699301633E-01
19	-9.2128743097707640E-03	43	5.1590021798482233E-01

20	-1.8830775873369020E-02	44	7.1080020379761377E-01
21	-2.7226498457701823E-02	45	8.8090632488444798E-01
22	-3.2022840857588906E-02	46	1.0068321641150089E+00
23	-3.0996332527754609E-02	47	1.0737914947736096E+00

4.6.13 Perceptual noise substitution (PNS)

4.6.13.1 Tool description

This tool is used to implement perceptual noise substitution coding within an ICS. Thus, certain sets of spectral coefficients are derived from random vectors rather than from Huffman coded symbols and an inverse quantization process. This is done selectively on a scalefactor band and group basis when perceptual noise substitution is flagged as active.

4.6.13.2 Definitions

hcod_sf[]	Huffman codeword from the Huffman code table used for coding of scalefactors (see subclause 4.6.2)
dpcm_noise_nrg[][]	Differentially encoded noise energy
noise_nrg[group][sfb]	Noise energy for each group and scalefactor band
spec[]	Array containing the channel spectrum of the respective channel

4.6.13.3 Decoding process

The use of the percpetual noise substitution tool is signaled by the use of the pseudo codebook NOISE_HCB (13).

Furthermore, if the same scalefactor band and group is coded by perceptual noise substitution in both channels of a channel pair, the correlation of the noise signal can be controlled by means of the ms_used field: While the default noise generation process works independently for each channel (separate generation of random vectors), the same random vector is used for both channels if ms_used[] is set for a particular scalefactor band and group or ms_mask_present is set to '10'. In this case, no M/S stereo coding is carried out (because M/S stereo coding and noise substitution coding are mutually exclusive). If the same scalefactor band and group is coded by perceptual noise substitution in only one channel of a channel pair the setting of ms_used[] is not evaluated.

The energy information for percpetual noise substitution decoding is represented by a "noise energy" value indicating the overall power of the substituted spectral coefficients in steps of 1.5 dB. If noise substitution coding is active for a particular group and scalefactor band, a noise energy value is transmitted instead of the scalefactor of the respective channel.

Noise energies are coded just like scalefactors, i.e. by Huffman coding of differential values:

· the start value for the DPCM decoding is given by global_gain.

· Differential decoding is done separately between scalefactors, intensity stereo positions and noise energies. In

other words, the noise energy decoder ignores interposed scalefactors and intensity stereo position values and

vice versa (see subclause 4.6.2.3.2)

The same codebook is used for coding of noise energies as for scalefactors.

One pseudo function is defined for use in perceptual noise substitution decoding:

```
function is_noise(group,sfb)  {
   1      for window groups / scalefactor bands with
             codebook sfb_cb[group][sfb] == NOISE_HCB
   0      otherwise
}
```

The noise substitution decoding process for one channel is defined by the following pseudo code:

```
nrg = global_gain - NOISE_OFFSET - 256;
for (g=0; g<num_window_groups; g++)  {
```

```
    /* Decode noise energies for this group */
    for (sfb=0; sfb<max_sfb; sfb++) {
        if (is_noise(g,sfb)) {
            nrg += dpcm_noise_nrg[g][sfb];
            noise_nrg[g][sfb] = nrg;
        }
    }
    /* Do perceptual noise substitution decoding */
    for (b=0; b<window_group_length[g]; b++)  {
        for (sfb=0; sfb<max_sfb; sfb++)  {
            if (is_noise(g,sfb))  {
                size = swb_offset[sfb+1] - swb_offset[sfb];
                /* Generate random vector */
                gen_rand_vector( &spec[g][b][sfb][0], size );
                nrg=0;
                for (i=0; i<size; i++) {
                    nrg+= spec[g][b][sfb][i] * spec[g][b][sfb][i];
                }
                sqrt_nrg = sqrt (nrg);
                scale *= 2.0^(0.25*noise_nrg [g][sfb]) / sqrt_nrg;
                /* scale random vector to desired target energy */
                for (i=0; i<size; i++) {
                    spec[g][b][sfb][i] *= scale;
                }
            }
        }
    }
}
```

The constant NOISE_OFFSET is used to adapt the range of average noise energy values to the usual range of scalefactors and has a value of 90.

The function gen_rand_vector(addr, size) generates a vector of length <size> with signed random values whereas their sum of squares is unequal to zero. A suitable random number generator can be realized using one multiplication/accumulation per random value.

In case of reversible variable length coding (RVLC) the start value for the DPCM backwards decoding is given by reversible_global_gain. The decoding of the noise energies is defined by the following pseudo code:

```
nrg = rev_global_gain-NOISE_OFFSET-256+dpcm_noise_last_position;
for (g = win-1; g >= 0; g--) {
  for (sfb = sfbmax-1; sfb >= 0; sfb--) {
    noise_nrg[g][sfb]=nrg;
    nrg -= dpcm_noise_nrg[g][sfb];
  }
}
```

4.6.13.4 Integration with the intra channel prediction tools

For scalefactor bands coded using PNS the corresponding predictors are switched to „off„, thus effectively overriding the status specified by the prediction_used mask. In addition, for scalefactor bands coded by perceptual noise substitution the predictors belonging to the corresponding spectral coefficients are reset (see ISO/IEC13818-7 subclause 8.3.3). The update of these predictors is done by feeding a value of zero as the "last quantized value" $x_{rec(n-1)}$.

If both Long Term Prediction and PNS are active for a particular scalefactor band and group, PNS takes precedence, i.e. the spectral coefficients in this scalefactor band are produced by the PNS tool only.

4.6.13.5 Integration with other AAC tools

The following interactions between the perceptual noise substitution tool and other AAC tools take place:

- Definition of a new pseudo Huffman codebook number NOISE_HCB = 13

- During Huffman decoding of the quantized spectral coefficients, the Huffman codebook table NOISE_HCB is treated exactly like the zero codebook ZERO_HCB, i.e. no Huffman codewords are read for the corresponding scalefactor band and group.
- If the same scalefactor band and group is coded by perceptual noise substitution in both channels of a channel pair, no M/S stereo decoding is carried out for this scalefactor band and group .
- The pseudo noise components generated by the perceptual noise substitution tool are injected into the output spectrum prior to the temporal noise shaping (TNS) processing step.

4.6.13.6 Integration into a scalable AAC-based coder (AudioObjectType AAC scalable)

The following rules apply for usage of the perceptual noise substitution tool in a scalable AAC-based coder:

- If a particular scalefactor band is coded by perceptual noise substitution in layer N, it only contributes to the output spectrum of the combined layers N and N+1 if all of the following requirements are fullfilled (see subclause 4.5.2.2.7):
  - layers N and N+1 are both either mono or stereo layers
  - layer N+1 does not use Intensity Stereo in this scalefactor band
  - layer N+1 does not use PNS in this scalefactor band
  - all spectral coefficients of layer N+1 in this scalefactor band are decoded to zero. In case of M/S coding this is required for both channels of the channel pair element.
- If a particular scalefactor band and group is coded by perceptual noise substitution in both channels of a channel pair, the higher (enhancement) layers may still use the M/S stereo flag ms_used[][] to signal the use of M/S stereo decoding.

4.6.14 Frequency selective switch (FSS) Module

The Frequency Selective Switching Unit (FSS) is used in various scalable coder configurations (see the diagrams in Figure 4.4 to Figure 4.15). It consists of a bank of switches which have the function to select one of two input signals, independently for each scale factor band (sfb). For each sfb a control bit is available which controls the selection. The form of the transmission of these control bits differs depending on the configuration in which the FSS module is used.

4.6.14.1 FSS in combined TwinVQ /CELP- AAC systems

4.6.14.1.1 Definitions

dc_group	Four consecutive scalefactor bands if the window type is not SHORT_WINDOW. One band of diff_short_lines, if the window type is SHORT_WINDOW.
no_of_dc_groups	If the window type is not SHORT_WINDOW, the number of groups depending on the sampling frequency is given in Table 4.112 below. If the window type is SHORT_WINDOW, no_of_dc_groups is '1'.
diff_short_lines	Only used, if the window type is SHORT_WINDOW. The number of spectral lines in the single dc_group per window, depending on the sampling rate, is given in Table 4.112 below.
diff_control_sfb[w][sfb]	Only applies, if the window type is not SHORT_WINDOW; These are the decoded **diff_control[w][dc_group]** values; One control bit for each switched scale factor band is available.

4.6.14.1.2 Decoding for the CELP – AAC combination

In the bitstream payload **diff_control[w][dc_group]** is used to transmit the huffman encoded values of diff_control_sfb[w][sfb] according to the following table:

Table 4.146 – Huffman code table for diff_control[w][dc_group]

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Code	0	20	21	22	23	24	25	8	9	26	27	28	29	30	31	1

Length	2	5	5	5	5	5	5	4	4	5	5	5	5	5	5	2

Table 4.112 expands the sampling frequency mapping specified in Table 4.55 for the CELP – AAC combination. In addition to the specification of the samping frequency dependent tables, it specifies the ratio between AAC sampling frequency and core sampling frequency, the values of no_of_dc_groups and diff_short_lines. In addition to the sampling frequency ratio, the target sampling frequencies for both, the core and the AAC enhancement layers are given as a reference.

Table 4.112 – Sampling frequency mapping for the AAC CELP – AAC combinations

			narrowband CELP				wideband CELP			
AAC sampling frequency range (Hz)	use tables for sampling frequenc (in kHz)	AAC target sampling frequency (kHz)	sampling frequency ratio between AAC and core	core target sampling frequency (kHz)	no_of_dc_groups	diff_short_lines	sampling frequency ratio between AAC and core	core target sampling frequency (kHz)	no_of_dc_groups	diff_short_lines
$fs_{\_aac}$ >=184035	96	192					12	16	4	8
184035> $fs_{\_aac}$ >=150264	96	176.4					12	14.7	4	8
150264> $fs_{\_aac}$ >=110851	96	128					8	16	5	13
110851> $fs_{\_aac}$ >=92017	96	96	12	8	4	8	6	16	6	18
92017> $fs_{\_aac}$ >=75132	88.2	88.2	12	7.35	4	8	6	14.7	6	18
75132> $fs_{\_aac}$ >=55426	64	64	8	8	5	13	4	16	6	26
55426> $fs_{\_aac}$ >=46009	48	48	6	8	5	18	3	16	7	36
46009> $fs_{\_aac}$ >=37566	44.1	44.1	6	7.35	5	18	3	14.7	7	36
37566> $fs_{\_aac}$ >=27713	32	32	4	8	6	26	2	16	8	54
27713> $fs_{\_aac}$ >=23004	24	24	3	8	8	36	1	24	9	104
23004> $fs_{\_aac}$ >=18783	22.05	22.05	3	7.35	8	36	1	22.05	9	104
18783> $fs_{\_aac}$ >=13856	16	16	2	8	8	54	1	16	10	104
13856> $fs_{\_aac}$ >=11502	12	12	1	12	10	104				
11502> $fs_{\_aac}$ >=9391	11.025	11.025	1	11.025	10	104				
9391> $fs_{\_aac}$	8	8	1	8	9	104				

4.6.14.1.3 FSS for the TwinVQ – AAC combination

The same huffman encoding (Table 4.146) of the values of diff_control_sfb[w][sfb] which is defined for the CELP – AAC combination, is also used for combined TwinVQ – AAC systems. However, the value of no_of_dc_groups is calculated from the value of max_sfb of the last TwinVQ layer as follows:

```
no_of_dc_groups = int ((max_sfb + 3) / 4 )
```

The value for diff_short_lines is taken from the scalefactor band table for SHORT_WINDOW of the corresponding sampling rate:

```
diff_short_lines = swb_offset [max_sfb]
```

4.6.14.1.4 Common decoding process

Decoding if the window type is not SHORT_WINDOW:

After huffman decoding diff_control[w][dc_group] from the bitstream payload, the array diff_control_sfb[w][sfb] is generated according to:

```
if ( ! SHORT_WINDOW ) {
   dc_group = 0;
   while (dc_group < no_of_ dc_groups) {
       for (i = 0; i < 4; i++) {
           diff_control_sfb[0][dc_group*4+i] = diff_control[0][dc_group]  & 0x8;
           diff_control[0][dc_group] <<= 1;
       }
       dc_group++;
   }
}
```

For all scale factor bands which did not get a value assigned in diff_control_sfb[w][sfb] in the above procedure, diff_control_sfb[w][sfb] is set to '1';

Finally the switching for all scale factor bands is done according to:

```
if (diff_control_sfb[w][sfb] == 0) {
   spectrum_out[w][sfb] = spectrum_AAC[w][sfb] + spectrum_Celp/TwinVQ[w][sfb];
} else {
   spectrum_out[w][sfb] = spectrum_AAC[w][sfb];
}
```

Decoding if the window type is SHORT_WINDOW:

If the window type is SHORT_WINDOW, there is only one band of diff_short_lines per window where the diff control mechanism is applied:

For the spectral lines from 0 to diff_short_lines-1:

```
if (diff_control_sfb[w][0] == 0) {
   spectrum_out[w] = spectrum_AAC [w] + spectrum_Celp/TwinVQ[w];
} else {
   spectrum_out[w] = spectrum_AAC[w];
}
```

For the remaining lines the output of the switch is identical to the input:

```
   spectrum_out[w] = spectrum_AAC [w];
```

4.6.14.2 FSS in combined mono / stereo scalable configurations

4.6.14.2.1 Decoding process

In a combined mono-stereo coder, where a mono (M) signal which is derived from a stereo input is coded with one or more mono layers, and later coded with one or more stereo layers, the FSS tool is also used to control the addition of the output signal of the combined M-coding stages to the Left (L) or Right (R) channel signals of the stereo coding stages. In this case the number of processed bands in the current layer is equal to **max_sfb** of the

current layer. However, if last_max_sfb is larger than **max_sfb** of the current layer, the unused **diff_control_lr[w][sfb]** bits are preserved for a subsequent layer. The control bits for these FSS modules are directly available from the **diff_control_lr[w][sfb]** syntax elements. Since the combined L+M, or R+M signals in one scale factor band (sfb) are only needed in subsequent coding stages of the current layer, if MS-coding is not selected for this specific sfb, **diff_control_lr[w][sfb]** is only transmitted for a sfb for which MS-coding is not selected.

Decoding if the window type is not SHORT_WINDOW:

For all scale factor bands for which no value is transmitted **diff_control_lr[w][sfb]** is set to '1';

The switching for all scale factor bands is done according to:

```
if (diff_control_lr[w][sfb] == 0) {
   spectrum_L/R_out[w][sfb] = spectrum_L/R[w][sfb] + 2 * spectrum_M[w][sfb];
} else {
   spectrum_L/R_out[w][sfb] = spectrum_L/R[w][sfb];
}
```

Decoding if the window type is SHORT_WINDOW:

If the window type is SHORT_WINDOW, the value **diff_control_lr[win][0]** is used for all scale factor bands from 0 to last_max_sfb-1. For all other bands a value of '1' is used.

4.6.15 Upsampling filter tool

4.6.15.1 Tool description

The upsampling filter tool is used to adapt the sampling rate of the (CELP) core coder to the sampling rate of the time/frequency coder. The upsampling filter uses the MDCT filterbank of the AAC encoder. This filterbank is very similar to the IMDCT filterbank, which is used in the decoder. Both use the same window functions.

The filterbank takes a block of time samples of the core coder output and inserts an appropriate number of zeroes between these samples to generate a signal at the desired higher sampling rate. These up-sampled values are then delayed by the number of samples given by the data element coreCoderDelay in the GASpecificConfig() of the first enhancement layer, and then modulated with the same window function which is used for the IMDCT of this block. The window type and the window shape from the IMDCT are used (To save RAM in the decoder it is also possible to delay the core bitstream payload by an apropriate number of core frames instead of delaying the upsampled core signal). Each block of input samples is overlapped by 50% with the immediately preceding block. The transform input block length N is set to either 2048 (1920) or 256 (240) samples depending on the value of **frameLengthFlag**.

The output of the MDCT filterbank is connected to the FSS module, which only uses the output values in the FSS-bands. Since the upper FSS band doesn't exceed half of the lower sampling rate, aliasing effects are omitted.

4.6.15.2 Definitions

up-sampling-factor	ratio of T/F coder sampling rate and core coder sampling rate.
$x_{in\text{-}mdct\text{-}core}[i]$	temporal data field, which is used to hold the up-sampled input to the MDCT filterbank
$x_{out\text{-}core}[i]$	Output samples of the core decoder

4.6.15.3 Decoding process

The analysis window length N for the transform is a function of the syntax element **window_sequence** and the algorithmic context. It is derived in an identical way to the procedure described for the Filterbank and Blockswitching tool.

4.6.15.3.1 Upsampling by insertion of zeroes

The input to the filterbank is generated by :

$x_{in\text{-}mdct\text{-}core}[k] = 0$ for k = [0 : N/2-1]

$x_{in\text{-}mdct\text{-}core}[\text{up-sampling-factor}*i] = x_{out\text{-}core}[i] * \text{up-sampling-factor}$ for i = [0 : N/2/up-sampling-factor-1]

4.6.15.3.2 Windowing and block switching

The adaptation of the time-frequency resolution of the filterbank is done by shifting between transforms whose input lengths are either 2048 (1920) or 256 (240) samples, synchronously to the decoder IMDCT filterbank. The selection between the 2048/256 or the 1920/240 pairs is done depending on the value of **frameLengthFlag**.

The windowed time domain values can be calculated by using exactly the same windows w(n) as defined for the IMDCT filterbank (see subclause 4.6.11).

The windowed coefficients are calculated by

$$z_{i,n} = w(n) \cdot x'_{\text{in\_mdct\_core}}(n);$$

4.6.15.3.3 MDCT

The windowed coefficients are transformed into the frequency domain with an MDCT.

The MDCT spectral coefficient, $X_{i,k}$, are defined as follows:

$$X_{i,k} = 2 \cdot \sum_{n=0}^{N-1} z_{i,n} \cos\left(\frac{2\pi}{N}\left(n + n_0\right)\left(k + \frac{1}{2}\right)\right) \text{ for } 0 \le k < N/2 .$$

where:

z_{in} = windowed input sequence

n = sample index

k = spectral coefficient index

i = block index

N = window length of the one transform window based on the window_ sequence value

n_0 = $(N/2 + 1)/2$

Only the output values from 0 to N/2/up-sampling-factor-1 can be used without aliasing distortions. This is ensured by a subsequently following FSS module.

4.6.16 Tools for AAC error resilience

4.6.16.1 Virtual codebooks for AAC section data

4.6.16.1.1 Tool description

Virtual codebooks are used to limit the largest absolute value permitted within a certain scale factor band where escape values are allowed, i. e. where codebook 11 is used originally. This tool allows 17 different codebook indices (11, 16...31) for the escape codebook. All these codebook indices refer to codebook 11. They are therefore called virtual codebooks. The difference between these codebook indices is the allowed maximum of spectral values belonging to the appropriate section. Due to this, errors within spectral data resulting in too large spectral values can be located and the according spectral lines can be concealed.

4.6.16.1.2 Decoding process

See subclause 4.5.2.3.2.

4.6.16.2 RVLC for AAC scalefactors

4.6.16.2.1 Tool description

RVLC (reversible variable length coding) is used instead of Huffman coding to achieve entropy coding of the scalefactors, because of its better performance in terms of error resilience. It can be considered to be a plug-in of the noiseless coding tool defined in 4.6.3 which allows decoding error resilient encoded scalefactor data.

RVLC enables additional backward decoding. Some error detection is possible in addition because not all nodes of the coding tree are used as codewords. The error resilience performance of the RVLC is as better as smaller the number of codewords. Therefore the RVLC table contains only values from -7 to +7, whereas the original Huffman codebook contains values from -60 to +60. A decoded value of ±7 is used as ESC_FLAG. It signals that an escape value exists, that has to be added to +7 or subtracted from -7 in order to find the actual scalefactor value. This escape value is Huffman encoded.

It is necessary to transmit an additional value in order to have a starting point for backward decoding for the DPCM encoded scalefactors. This value is called reversible global gain. If intensity stereo coding or PNS is used, additional values are also necessary for them. The length of the RVLC bitstream payload part has to be transmitted to allow backward decoding. Furthermore the length of the bitstream payload part containing the escape codewords should be transmitted to keep synchronization in case of bit errors.

4.6.16.2.2 Definitions

The following data elements are available within the bitstream payload, if the GASpecificConfig() enables the RVLC tool.

sf_concealment: is a data field that indicates the similarity between scale factors of the last frame and those of the current one. It shall be set to '0', if the scale factors of the last frame are dissimilar to those of the current frame. It shall set to '1', if they are similar.
Note: This data field is not required to decode error-free payload, but might be used to apply appropriate concealment strategies in case of corrupted scale factor data. No similarity criterion is specified since concealment is out of the scope of this standard.

rev_global_gain contains the last scalefactor value as a start value for the backward decoding. The length of this data field is 8 bits.

length_of_rvlc_sf is a data field that contains the length of the current RVLC data part in bits, including the DPCM start value for PNS. The length of this data field depends on window_sequence: If window_sequence == EIGHT_SHORT_SEQUENCE, the field consists of 11 bits, otherwise it consists of 9 bits.

rvlc_cod_sf RVLC word from the RVLC table used for coding of scalefactors, intensity positions or noise energy.

sf_escapes_present is a data field that signals whether there are escapes coded in the bitstream payload or not. The length of this data is 1 bit.

length_of_rvlc_escapes is a data field that contains the length of the current RVLC escape data part in bits. The length of this data is 8 bits.

rvlc_esc_sf Huffman codeword from the Huffman table for RVLC-ESC-values used for coding values larger than ±6.

dpcm_is_last_position DPCM value allowing backward decoding of intensity stereo data part. It is the symmetric value to dpcm_is_position.

dpcm_noise_last_position DPCM value allowing backward decoding of PNS data part. The length of this data is 9 bit. It is the symmetric value to dpcm_noise_nrg.

4.6.16.2.3 Decoding process

See subclause 4.6.2.3.2.

4.6.16.2.4 Tables

Table 4.147 – RVLC codebook

index	length	codeword
-7	7	65
-6	9	257
-5	8	129
-4	6	33
-3	5	17
-2	4	9
-1	3	5
0	1	0
1	3	7
2	5	27
3	6	51
4	7	107
5	8	195
6	9	427
7	7	99

Table 4.148 – Asymmetric (forbidden) codewords

length	codeword
6	50
7	96
9	256
8	194
7	98
6	52
9	426
8	212

Table 4.149 – Huffman codebook for RVLC escape values

index	length	codeword
0	2	2
1	2	0
2	3	6
3	3	2
4	4	14
5	5	31
6	5	15
7	5	13
8	6	61
9	6	29
10	6	25
11	6	24
12	7	120
13	7	56
14	8	242
15	8	114
16	9	486
17	9	230
18	10	974
19	10	463
20	11	1950
21	11	1951

index	length	codeword
27	20	473482
28	20	473483
29	20	473484
30	20	473485
31	20	473486
32	20	473487
33	20	473488
34	20	473489
35	20	473490
36	20	473491
37	20	473492
38	20	473493
39	20	473494
40	20	473495
41	20	473496
42	20	473497
43	20	473498
44	20	473499
45	20	473500
46	20	473501
47	20	473502
48	20	473503

22	11	925	49	19	236736
23	12	1848	50	19	236737
24	14	7399	51	19	236738
25	13	3698	52	19	236739
26	15	14797	53	19	236740

4.6.16.3 Huffman codeword reordering (HCR) for AAC spectral data

4.6.16.3.1 Tool description

The Huffman codeword reordering (HCR) algorithm for AAC spectral data is based on the fact that some of the codewords can be placed at known positions so that these codewords can be decoded independent of any error within other codewords. Therefore, this algorithm avoids error propagation to those codewords, the so-called priority codewords (PCW). To achieve this, segments of known length are defined and those codewords are placed at the beginning of these segments.

The remaining codewords (non-priority codewords, non-PCW) are filled into the gaps left by the PCWs using a special algorithm that minimizes error propagation to the non-PCWs codewords.

This reordering algorithm does not increase the size of spectral data.

Before applying the reordering algorithm itself, a pre-sorting process is applied to the codewords. It sorts all codewords depending on their importance, i. e. it determines the PCWs.

4.6.16.3.2 Definitions

The following data elements are available within the bitstream payload, if the GASpecificConfig() enables the HCR tool.

length_of_reordered_spectral_data is a 14-bit data field that contains the length of spectral data in bits. The maximum value is 6144 in case of a single_channel_element(), a coupling_channel_element() and a lfe_channel_element() and 12288 in case of a channel_pair_element(). Larger values are reserved for future use. If those values occur, current decoders have to replace them by the valid maximum value.

length_of_longest_codeword is a 6-bit data field that contains the length of the longest codeword available within the current spectral data in bits. This field is used to decrease the distance between protected codewords. Valid values are between 0 and 49. Values between 50 and 63 are reserved for future use. If those values occur, current decoders have to replace them by 49.

4.6.16.3.3 Bitstream payload structure

4.6.16.3.3.1 Pre-sorting

Subclause 4.5.2.3.5 is not valid if this tool is used. Instead, the procedure described in the following paragraphs has to be applied:

For explanation of the pre-sorting steps the term “unit” is introduced. A unit covers four spectral lines, i. e. two two-dimensional codewords or one four-dimensional codeword. In case of two two-dimensional codewords their natural order is kept, i. e. the higher frequency codeword follows the lower frequency codeword.

In case of one long window (1024 spectral lines per long block, one long block per frame), each window contains 256 units.

In case of eight short windows (128 spectral lines per short block, eight short blocks per frame), each window contains 32 units.

First pre-sorting step:

Units representing the same part of the spectrum are collected together in temporal order and denoted as unit group. In case of one long window, each unit group contains one unit. In case of eight short windows, each unit group contains eight units.

Unit groups are collected ascending in spectral direction. For one long window, that gives the original codeword order, but for eight short windows a unit based window interleaving has been applied.

Using this scheme, the codewords representing the lowest frequencies are the first codewords within spectral data for both, long and short blocks.

Table 4.150 shows an example output of the first pre-sorting step for short blocks, assuming two-dimensional codebooks for window 0, 1, 6, and 7 and four-dimensional codebooks for window 2, 3, 4, and 5.

Second pre-sorting step:

The more energy a spectral line contains, the more audible is its distortion. The energy within spectral lines is related to the used codebook. Codebooks with low numbers can represent only low values and allow only small errors, while codebooks with high numbers can represent high values and allow large errors.

Therefore, the codewords are pre-sorted depending on the used codebook. If the error resilient section data is used, the order is 11, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 9/10, 7/8, 5/6, 3/4, 1/2. If the normal section data is used, the order is 11, 9/10, 7/8, 5/6, 3/4, 1/2. This order is based on the largest absolute value of the tables. This second pre-sorting step is done on the described unit by unit base used in the first pre-sorting step. The output of the first pre-sorting step is scanned in a consecutive way for each codebook.

Assigning numbers to units according the following metric can perform these two pre-sorting steps:

```
codbookPriority[32] =
{x,21,21,20,20,19,19,18,18,17,17,0,x,x,x,x,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1}

assignedUnitNr = ( codebookPriority[cb] * maxNrOfLinesInWindow
                          + nrOfFirstLineInUnit ) * MaxNrOfWindows + window
```

with:

codebookPriority[cb]	codebook priority according the second pre-sorting step.
maxNrOfLinesInWindow	constant number: f in case of one long window and 128 in case of eight short windows
nrOfFirstLineInUnit	a number between 0 and 1020 in case of one long window and between 0 and 124 in case of eight short windows (this number is always a multiple of four)
maxNrOfWindows	constant number: 1 in case of one long window and 8 in case of eight short windows
window	always 0 in case of one long window, a number between 0 and 7 in case of eight short windows and sort the units in ascending order using these assigned unit numbers.

Encoder note: In order to reduce audible artifacts in case of errors within spectral data it is strongly recommended to use codebook 11 only if necessary!

4.6.16.3.3.2 Segment width and segment instantiation

The segment widths depend on the Huffman codebook used. They are derived as the minimums of the (codebook dependent) maximum codeword length and the transmitted longest codeword length:

```
segmentWidth = min (maxCwLen, length_of_longest_codeword)
```

Table 4.151 shows the values of maxCwLen depending on the Huffman codebook.

Segments are instantiated until the available buffer is exhausted, whereas the size of this buffer is given by the data element length_of_reordered_spectral_data. The remaining bits at the end of the buffer increase the size of the last segment.

4.6.16.3.3.3 Order of Huffman codewords in spectral data

Figure 4.34 shows the general scheme of the segmentation and the arrangement of the PCWs. In this Figure 4.34, five segments can be provided to protect codewords from section 0 and section 1 against error propagation.

Segment widths are different, because the length of the longest possible codeword depends on the current codebook.

The writing scheme for the non-PCWs is as follows (PCWs have been written already):

The proposed scheme introduces the term set. A set contains a certain number of codewords. Given N is the number of segments; all sets except the last one contain N codewords. Non-PCWs are written consecutively into these sets. Due to the pre-sorting algorithm set one contains the most important non-PCWs. The importance of the codewords stored within a set is the smaller the higher the set number.

Sets are written consecutively. If one set was written completely, writing of the next set starts. To improve the error propagation behavior between consecutive sets, the writing direction within segments changes from set to set. While PCWs are written from left to right, codewords of set one are written from right to left, codewords of set two are again written from left to right and so on. Writing into a remaining part of a segment always starts at the outermost position of the remaining part of that segment (leftmost position for write direction from left to right and rightmost position for write direction from right to left).

Writing of a set might need several trials.

First trial: The first codeword of the current set is written into the remaining part of the first segment, the second codeword into the remaining part of the second segment and so on. The last codeword of the current set is written into the remaining part of the last segment.

Second trial: The remaining part of the first codeword (if any) is written into the remaining part of the second segment, the remaining part of the second codeword (if any) into the remaining part of the third segment and so on. The remaining part of the last codeword (if any) is written into the remaining part of the first segment (modulo shift).

If a codeword does not fit into the remaining part of a segment, it is only partly written and its remaining part is stored. At least after a maximum of N trials all codewords are completely written into segments.

4.6.16.3.3.4 Encoding process

The structure of the reordered spectral data cannot be described within the C like syntax commonly used. Therefore, Figure 4.35 shows an example and the following c-like description is provided:

```
/* helper functions */
void InitReordering(void);
  /* Initializes variables used by the reordering functions like the segment
     widths and the used offsets in segments and codewords. */

void InitRemainingBitsInCodeword(void);
  /* Initializes remainingBitsInCodeword[] array for each codeword with
     the total size of the codeword. */

int  WriteCodewordToSegment(codewordNr, segmentNr, direction);
  /* Writes a codeword or only a part of a codeword indexed by codewordNr
     to the segment indexed by segmentNr with a given direction.
     Write offsets for each segment are handled internally.
     The function returns the number of bits written to the segment.
     This number may be lower than the codeword length.
     WriteCodewordToSegment handles already written parts of the codeword
     internally. */

void ToggleWriteDirection(void);
  /* Toggles the write direction in the segments between forward and backward. */

/* (input) variables */
numberOfCodewords;  /* 15 in the example */
numberOfSegments;   /*  6 in the example */
numberOfSets;       /*  3 in the example */

ReorderSpectralData()
{
  InitReordering();
  InitRemainingBitsInCodeword();

  /* first step: write PCWs (set 0) */
  writeDirection = forward;
  for (codeword = 0; codeword < numberOfSegments; codeword ++) {
```

```
    WriteCodewordToSegment(codeword, codeword, writeDirection);
  }

  /* second step: write nonPCWs */
  for (set = 1; set < numberOfSets; set++) {
    ToggleWriteDirection();
    for (trial = 0; trial < numberOfSegments; trial++) {
      for (codewordBase = 0; codewordBase < numberOfSegments; codewordBase++) {
        segment  = (trial + codewordBase) % numberOfSegments;
        codeword = codewordBase + set*numberOfSegments;

        if (remainingBitsInCodeword[codeword] > 0) {
          remainingBitsInCodeword[codeword] -= WriteCodewordToSegment(codeword,
                                                                      segment,
                                                                      writeDirection);
        }
      }
    }
  }
}
```

4.6.16.3.4 Decoding process

See subclauses 4.5.2.3.2 and 4.6.3.3.

4.6.16.3.5 Tables

Table 4.150 – Example output of the first pre-sorting step for short blocks, assuming two-dimensional codebooks for window 0, 1, 6, and 7 and four-dimensional codebooks for window 2, 3, 4, and 5

index	codeword entry	
	window	window index
0	0	0
1	0	1
2	1	0
3	1	1
4	2	0
5	3	0
6	4	0
7	5	0
8	6	0
9	6	1
10	7	0
11	7	1
12	0	2
13	0	3
14	1	2
15	1	3
16	2	1
17	3	1
18	4	1
19	5	1
20	6	2
21	6	3
22	7	2
23	7	3
…	…	…

Table 4.151 – Values of maxCwLen depending on the Huffman codebook

codebook	maximum codeword length (maxCwLen)
0	0
1	11
2	9
3	20
4	16
5	13
6	11
7	14
8	12
9	17
10	14
11	49
16	14
17	17
18	21
19	21
20	25
21	25
22	29
23	29
24	29
25	29
26	33
27	33
28	33
29	37
30	37
31	41

4.6.16.3.6 Figures

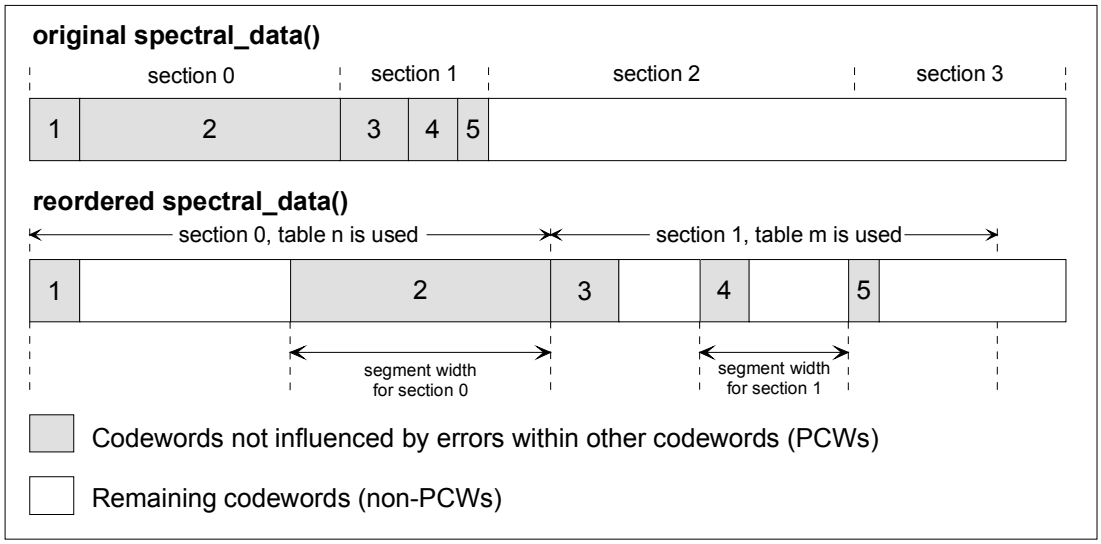


Figure 4.34 – General scheme of segmentation and arrangement of PCWs

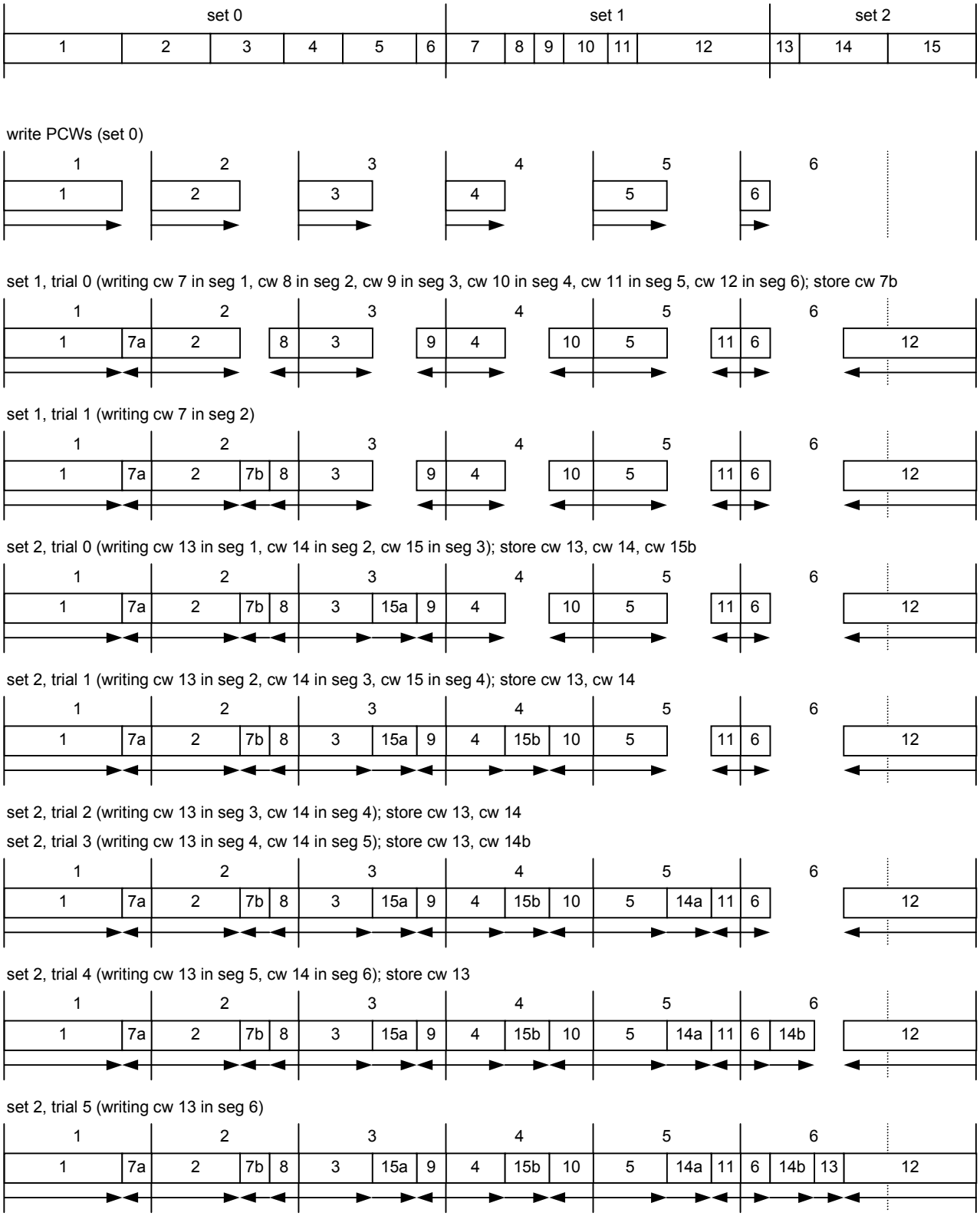


Figure 4.35 – Example for HCR encoding algorithm (only one segment width, pre-sorting has been done before)

4.6.17 Low delay codec

4.6.17.1 Introduction

The low delay coding functionality provides the ability to extend the usage of generic low bitrate audio coding to applications requiring a very low delay of the encoding / decoding chain (e.g. full-duplex real-time communications).

This subclause specifies a low delay audio coder providing a mode with an algorithmic delay not exceeding 20 ms.

The overall algorithmic delay of a general audio coder is determined by the following factors:

- **Frame length**
  For block-based processing, a certain amount of time has to pass to collect the samples belonging to one block
- **Filterbank delay**
  Use of an analysis-synthesis filterbank pair causes a certain amount of delay.
- **Look-ahead for block switching decision**
  Due to the underlying principles of the block switching scheme, the detection of transients has to use a certain amount of "look-ahead" in order to ensure that all transient signal parts are covered properly by short windows.
- **Use of bit reservoir**
  While the bit reservoir facilitates the use of a locally varying bitrate, this implies an additional delay depending on the size of the bit reservoir relative to the average bitrate per block.

The overall algorithmic delay can be calculated as

$$t_{delay} = \frac{N_{Frame} + N_{FB} + N_{look\_ahead} + N_{bitres}}{F_s}$$

where F_S is the coder sampling rate, N_{Frame} is the frame size, N_{FB} is the delay due to the filterbank (s), N_{look_ahead} corresponds to the look-ahead delay for block switching and N_{bitres} is the delay due to the use of the bit reservoir.

The low delay codec is derived from the AAC LTP audio object type, i.e. a coder consisting of the low complexity AAC codec plus the PNS (Perceptual Noise Substitution) and the LTP (Long Term Predictor) tools. Figure 4.36 shows the overall structure of the decoder.

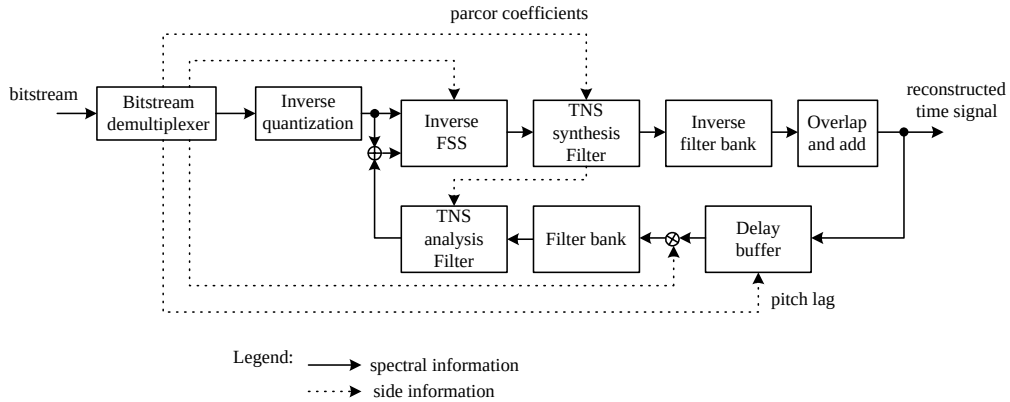


Figure 4.36 – Decoder structure

Figure 4.37 shows the overall structure of the encoder.

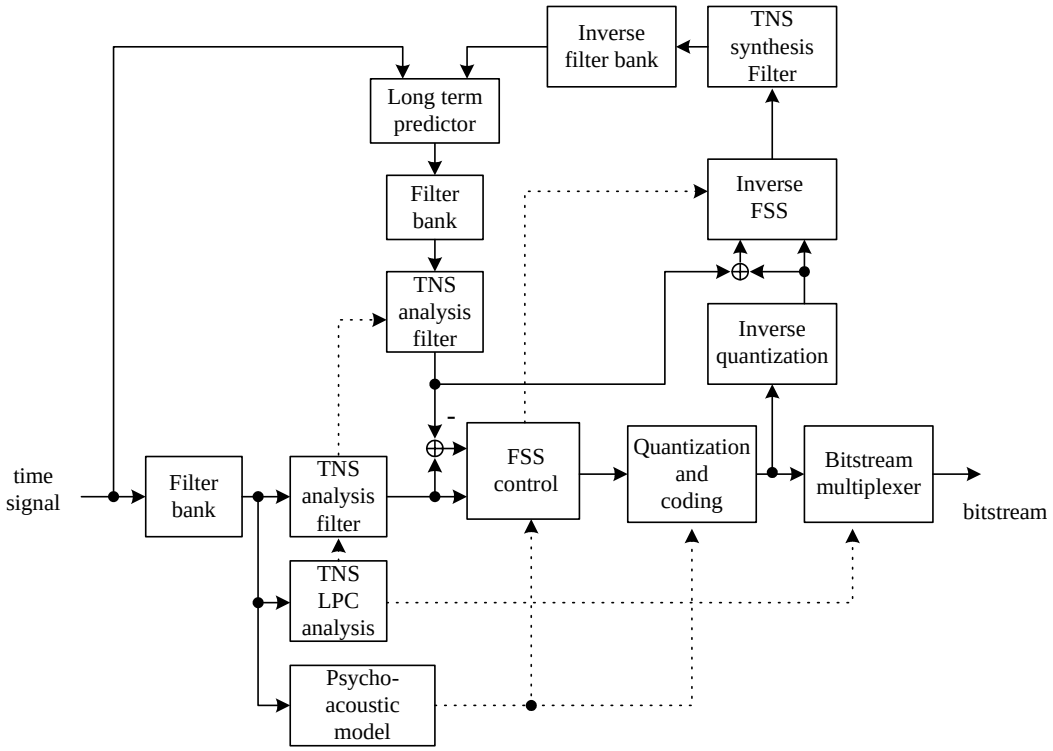


Figure 4.37 – Encoder structure

4.6.17.2 Coder description

The low delay codec is defined by the following modifications with respect to the standard algorithm (i.e. AAC LTP audio object type) to achieve low delay operation:

4.6.17.2.1 Frame size/window length

The length of the analysis window is reduced to 1024 or 960 time domain samples corresponding to 512 and 480 spectral values, respectively. The latter choice enables the coder to have a frame size that is commensurate with widely used speech codecs (20 ms). The corresponding scalefactor band tables are given in subclause 4.5.4.

4.6.17.2.2 Block switching

Due to the contribution of the look-ahead time to the overall delay, no block switching is used.

4.6.17.2.3 Window shape

As stated in the previous chapter, block switching is not used in the low delay coder to keep the delay as low as possible. As an alternative tool to improve coding of transient signals, the low delay coder uses the window shape switching feature with a slight modification compared to normal-delay AAC: The low delay coder still uses the sine window shape, but the Kaiser-Bessel derived window is replaced by a low-overlap window. As indicated by its name, this window has a rather low overlap with the following window, thus being optimized for the use of the TNS tool to prevent preecho artefacts in case of transient signals. For normal coding of non-transient signals the sine window is used because of its advantageous frequency response.

In line with normal-delay AAC, the window_shape indicates the shape of the trailing part (i.e. the second half) of the analysis window. The shape of the leading part (i.e. the first half) of the analysis window is identical to the window_shape of the last block.

Table 4.152 – window depending on window_shape

window_shape	window
0x0	sine
0x1	low-overlap

The low-overlap window is defined by:

$$W(i) = \begin{cases} 0 & i = 0..3 \cdot N/16 - 1 \\ \sin\left[\frac{\pi(i - 3 \cdot N/16 + 0.5)}{N/4}\right] & i = 3 \cdot N/16..5 \cdot N/16 - 1 \\ 1 & i = 5 \cdot N/16..11 \cdot N/16 - 1 \\ \sin\left[\frac{\pi(i - 9 \cdot N/16 + 0.5)}{N/4}\right] & i = 11 \cdot N/16..13 \cdot N/16 - 1 \\ 0 & i = 13 \cdot N/16..N - 1 \end{cases}$$

with $N = 1024$ or $N = 960$.

4.6.17.2.4 Bit reservoir use

Use of the bit reservoir is minimized in order to reach the desired target delay. As one extreme case, no bit reservoir is used at all.

4.6.17.2.5 Tables for temporal noise shaping (TNS)

The following tables specify the value of TNS_MAX_BANDS for the low delay coder:

Table 4.153 – TNS_MAX_BANDS in case of 480 samples per frame

Sampling Rate	TNS_MAX_BANDS
48000	31
44100	32
32000	37
24000	30
22050	30

Table 4.154 – TNS_MAX_BANDS in case of 512 samples per frame

Sampling Rate	TNS_MAX_BANDS
48000	31
44100	32
32000	37
24000	31
22050	31

4.6.17.2.6 Long term prediction

The size of the LTP delay buffer size is scaled down proportionally with the frame size. Thus, the size is 2048 and 1920 samples for frame sizes of N=512 and N=480, respectively (see subclause 4.6.7).

4.6.17.2.7 Adaptation to systems using lower sampling rates

In certain applications it may be necessary to integrate the low delay decoder into an audio system running at lower sampling rates (e.g. 16 kHz) while the nominal sampling rate of the bitstream payload is much higher (e.g. 48 kHz, corresponding to an algorithmic codec delay of approx. 20 ms). In such cases, it is favorable to decode the output of the low delay codec directly at the target sampling rate rather than using an additional sampling rate conversion operation after decoding.

This can be approximated by appropriate downscaling of both, the frame size and the sampling rate, by some integer factor (e.g. 2, 3), resulting in the same time/frequency resolution of the codec. For example, the codec output can be generated at 16 kHz sampling rate instead of the nominal 48 kHz by retaining only the lowest third

(i.e. 480/3 = 160) of the spectral coefficients prior to the synthesis filterbank and reducing the inverse transform size to one third (i.e. window size 960/3 = 320).

As a consequence, decoding for lower sampling rates reduces both memory and computational requirements, but may not produce exactly the same output as a full-bandwidth decoding, followed by band limiting and sample rate conversion.

Please note that decoding at a lower sampling rate, as described above, does not affect the interpretation of levels, which refers to the nominal sampling rate of the AAC low delay bitstream payload.

4.6.18 SBR tool

4.6.18.1 Tool description

The human voice and musical instruments generate either quasi-stationary excitation signals that emerge from oscillating systems or signals originated from different noise sources. A wide-band excitation spectrum could be initialized by one or by a set of several sources, e.g. vocal cords, strings, and reeds etc. They all have different frequency components depending on the source. The excitation signals are subsequently filtered by resonators such as the vocal tract, violin body etc, giving the voice or musical instrument its characteristic tone color or timbre. A bandwidth limitation of such a signal is equivalent to a truncation of the sequence of harmonics. Such a truncation alters the perceived timbre and the audio signal sounds “muffled” or “dull”, and particularly for speech the intelligibility may be reduced.

The SBR tool (Spectral Band Replication) extends the audio bandwidth of the decoded bandwidth-limited audio signal. The process is based on replication of the sequences of harmonics, previously truncated in order to reduce data rate, from the available bandwidth limited signal and control data obtained from the encoder. The ratio between tonal and noise-like components is maintained by adaptive inverse filtering as well as optional addition of noise and sinusoidals.

4.6.18.2 Definitions

4.6.18.2.1 SBR specific definitions

For the purpose of sublause 4.6.18, the following terms and definitions apply.

4.6.18.2.1.1

band
(as in limiter band, noise floor band, etc.) a group of consecutive QMF subbands

4.6.18.2.1.2

chirp factor
the bandwidth expansion factor of the formants described by a LPC polynomial

4.6.18.2.1.3

Down Sampled SBR
the SBR Tool with a modified synthesis filterbank resulting in a down sampled output signal with the same sample rate as the input signal to the SBR Tool. May be used whenever a lower sample rate output is desired.

4.6.18.2.1.4

envelope scalefactor
an element representing the averaged energy of a signal over a region described by a frequency band and a time segment

4.6.18.2.1.5

frequency band
interval in frequency, group of consecutive QMF subbands

4.6.18.2.1.6

frequency border

frequency band delimiter, expressed as a specific QMF subband

4.6.18.2.1.7

NA
Not Applicable

4.6.18.2.1.8

noise floor
a vector of noise floor scalefactors

4.6.18.2.1.9

noise floor scalefactor
an element associated with a region described by a frequency band and a time segment, representing the ratio between the energy of the noise to be added to the envelope adjusted HF generated signal and the energy of the same

4.6.18.2.1.10

patch
a number of adjoining QMF subbands moved to a different frequency location

4.6.18.2.1.11

QMF
Quadrature Mirror Filter

4.6.18.2.1.12

SBR
Spectral Band Replication

4.6.18.2.1.13

SBR envelope
a vector of envelope scalefactors

4.6.18.2.1.14

SBR frame
time segment associated with one SBR extension data element

4.6.18.2.1.15

SBR range
the frequency range of the signal generated by the SBR algorithm

4.6.18.2.1.16

subband
a frequency range represented by one row in a QMF matrix, carrying a subsampled signal

4.6.18.2.1.17

time border
time segment delimiter, expressed as a specific time slot

4.6.18.2.1.18

time segment
interval in time, group of consecutive time slots

4.6.18.2.1.19

time / frequency grid
a description of SBR envelope time segments and associated frequency resolution tables as well as description of noise floor time segments

4.6.18.2.1.20

time slot
finest resolution in time for SBR envelopes and noise floors. One time slot equals two subsamples in the QMF domain

4.6.18.2.2 SBR specific notation

The description of the SBR tool uses the following notation:

- Vectors are indicated by bold lower-case names, e.g. **vector**.
- Matrices (and vectors of vectors) are indicated by bold upper-case single letter names, e.g. $\mathbf{M}$.
- Variables are indicated by italic, e.g. *variable*.
- Functions are indicated as *func(x)*.
- Data elements are indicated as multiple-word names with prefix "bs_", e.g. bs_bitstream_element.

For equations in the text, normal mathematical interpretation is assumed (no rounding or truncation unless explicitly stated). Hence the following example from the text,

$$\mathbf{Q}_{Mapped}\left(m-k_x,l\right)=\mathbf{Q}_{Orig}\left(i,k\left(l\right)\right),\mathbf{f}_{TableNoise}(i)\le m<\mathbf{f}_{TableNoise}(i+1),0\le i<N_Q,0\le l<L_E$$

where $k(l)$ is defined by $\mathbf{t}(l)\ge\mathbf{t}_Q\left(k\left(l\right)\right),\mathbf{t}(l+1)\le\mathbf{t}_Q\left(k\left(l\right)+1\right)$

should be interpreted as follows. $\mathbf{Q}_{Mapped}\left(m-k_x,l\right)$ equals $\mathbf{Q}_{Orig}\left(i,k\left(l\right)\right)$ for $\mathbf{f}_{TableNoise}\left(i\right)\le m<\mathbf{f}_{TableNoise}\left(i+1\right)$ and $0\le i<N_Q$ and $0\le l<L_E$. The function $k(l)$ returns, for a given l, a value for which $\mathbf{t}(l)\ge\mathbf{t}_Q\left(k\left(l\right)\right)$ and $\mathbf{t}(l+1)\le\mathbf{t}_Q\left(k\left(l\right)+1\right)$ is true. The result is a $\mathbf{Q}_{Mapped}$ matrix that is piecewise constant.

The expression $\sum_{k=a}^{b} f\left(k\right)$ evaluates to zero if $b<a$.

For flowcharts, normal pseudo-code interpretation is assumed, with no rounding or truncation unless explicitly stated.

4.6.18.2.3 Scalar operations

X^* is the complex conjugate of X

4.6.18.2.4 Vector operations

$\mathbf{y}=sort\left(\mathbf{x}\right)$. $\mathbf{y}$ is equal to the sorted vector $\mathbf{x}$, where the elements of $\mathbf{x}$ are sorted in ascending order.

$y=length\left(\mathbf{x}\right)$. y is the number of elements of the vector $\mathbf{x}$.

$y=ceil\left(x\right)$ represents rounding to the nearest integer towards infinity.

4.6.18.2.5 Constants

$\varepsilon=1$	A constant to avoid division by zero, e.g. 96 dB below maximum signal input.
$HI = 1$	Index used for SBR envelope high frequency resolution.
$LO = 0$	Index used for SBR envelope low frequency resolution.
$NOISE\_FLOOR\_OFFSET = 6$	Offset of noise floor.

$RATE = 2$	A constant indicating the number of QMF subband samples per timeslot.

4.6.18.2.6 Variables

Description of variables defined in one subclause and used in other subclauses.

ch	is the current channel.
$bsco$	bandwidth scalable codec offset, indicates for a bandwidth scalable system the number of QMF subbands above k_x in the SBR range that should be replaced by AAC data from a bandwidth scalable enhancement layer. If a bandwidth scalable core coder is not used, the variable is zero.
$\mathbf{E}_{Orig}$	has L_E columns where each column is of length N_{Low} or N_{High} depending on the frequency resolution for each SBR envelope. The elements in $\mathbf{E}_{Orig}$ contains the envelope scalefactors of the original signal.
$\mathbf{F} = \left[\mathbf{f}_{TableLow}, \mathbf{f}_{TableHigh}\right]$	has two column vectors containing the frequency border tables for low and high frequency resolution.
Fs_{SBR}	internal sampling frequency of the SBR Tool, twice the sampling frequency of the core coder (after sampling frequency mapping, Table 4.68). The sampling frequency of the SBR enhanced output signal is equal to the internal sampling frequency of the SBR Tool, unless the SBR Tool is operated in downsampled mode. If the SBR Tool is operated in downsampled mode, the output sampling frequency is equal to the sampling frequency of the core coder.
$\mathbf{f}_{Master}$	is of length $N_{Master}+1$ and contains QMF master frequency grouping information.
$\mathbf{f}_{TableHigh}$	is of length $N_{High}+1$ and contains frequency borders for high frequency resolution SBR envelopes.
$\mathbf{f}_{TableLim}$	is of length N_L+1 and contains frequency borders used by the limiter.
$\mathbf{f}_{TableLow}$	is of length $N_{Low}+1$ and contains frequency borders for low frequency resolution SBR envelopes.
$\mathbf{f}_{TableNoise}$	is of length N_Q+1 and contains frequency borders used by noise floors.
k_x	the first QMF subband in the SBR range.
k_0	the first QMF subband in the $\mathbf{f}_{Master}$ table.
L_E	number of SBR envelopes.
L_Q	number of noise floors.
M	number of QMF subbands in the SBR range.
$middleBorder$	points to a specific time border.
N_L	number of limiter bands.
N_{Master}	number of frequency bands in the master frequency resolution table.
N_Q	number of noise floor bands.
$\mathbf{n} = [N_{Low}, N_{High}]$	number of frequency bands for low and high frequency resolution.
$numPatches$	a variable indicating the number of patches in the SBR range.
$numTimeSlots$	number of SBR envelope time slots that exist within an AAC frame, 16 for a 1024 AAC frame and 15 for a 960 AAC frame.
$\mathbf{panOffset} = \left[24,12\right]$	offset-values for the SBR envelope and noise floor data, when using coupled channels.

patchBorders	a vector containing the frequency borders of the patches.
patchNumSubbands	a vector holding the number of subbands in every patch.
$\mathbf{Q}_{Orig}$	has L_Q columns where each column is of length N_Q and contains the noise floor scalefactors.
$\mathbf{r} = [r_0, ..., r_{L-1}]$	frequency resolution for all SBR envelopes in the current SBR frame, zero for low resolution, one for high resolution.
reset	a variable in the encoder and the decoder set to one if certain data elements have changed from the previous SBR frame, otherwise set to zero.
$\mathbf{t}_E$	is of length L_E+1 and contains start and stop time borders for all SBR envelopes in the current SBR frame.
t_{HFAdj}	offset for the envelope adjuster module.
t_{HFGen}	offset for the HF-generation module.
$\mathbf{t}_Q$	is of length L_Q+1 and contains start and stop time borders for all noise floors in the current SBR frame.
$\mathbf{W}$	is the subband matrix where the QMF filtered subband samples are stored.
$\mathbf{X}_{High}$	is the complex input QMF bank subband matrix to the HF adjuster.
$\mathbf{X}_{Low}$	is the complex input QMF bank subband matrix to the HF generator.
$\mathbf{Y}$	is the complex output QMF bank subband matrix from the HF adjuster.

4.6.18.3 Decoding process

4.6.18.3.1 Introduction

SBR incorporates adaptive time and frequency resolution for the envelope coding and adjustment. The adaptation is obtained by flexible grouping of QMF subband samples in time and frequency. For each such group, a corresponding scalefactor is calculated and transmitted. The subclause 4.6.18.3 describes how to recreate the time and frequency grouping chosen by the encoder. Furthermore, it shows how delta coded SBR envelopes and noise floors are decoded. The decoding process is outlined for one single channel element as well as for one channel pair element. For a single channel element, the channel number is denoted zero. For channel pair elements, the two channels are indexed zero and one, where zero represents the firstly decoded channel data in the channel pair element and one represents the secondly decoded channel in the channel pair element. The system is reset (*reset = 1*) if the value of any of the following data elements in the SBR header differs from that of the previous SBR frame:

- *bs_start_freq*
- *bs_stop_freq*
- *bs_freq_scale*
- *bs_alter_scale*
- *bs_xover_band*
- *bs_noise_bands*

4.6.18.3.2 Frequency band tables

The grouping of QMF subband samples in frequency is described by frequency band tables. The tables are defined by functions, most arguments of which are transmitted in the SBR header. For each SBR envelope, two frequency band tables are available, a high frequency resolution table, $\mathbf{f}_{TableHigh}$, and a low frequency resolution table, $\mathbf{f}_{TableLow}$. The noise floor and the limiter also have corresponding frequency band tables, $\mathbf{f}_{TableNoise}$ and $\mathbf{f}_{TableLim}$. All aforementioned tables are derived from one master frequency band table, $\mathbf{f}_{Master}$. The frequency band tables contain the frequency borders for each frequency band, represented as QMF subbands. Each frequency band is

defined by a start frequency border and a stop frequency border. The QMF subband indicated by the start frequency border is included in the frequency band, the QMF subband indicated by the stop frequency border is excluded from the frequency band. For all tables derived from $\mathbf{f}_{Master}$ the stop frequency border of band n equals the start frequency border of band $n+1$, where n is an arbitrary frequency band in the table. The following subclauses describe how to calculate $\mathbf{f}_{Master}$, $\mathbf{f}_{TableHigh}$, $\mathbf{f}_{TableLow}$, $\mathbf{f}_{TableNoise}$ and $\mathbf{f}_{TableLim}$.

4.6.18.3.2.1 Master frequency band table

In order to build the master frequency band table, the QMF subbands representing the boundaries of the table must first be calculated. The subband representing the lower frequency boundary of the table is denoted k_0, and is defined by:

$$k_0 = startMin + \mathbf{offset}\left(bs\_start\_freq\right),$$

where

$$\mathbf{offset} = \begin{cases} [-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7] & , Fs_{SBR} = 16000 \\ [-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,9,11,13] & , Fs_{SBR} = 22050 \\ [-5,-3,-2,-1,0,1,2,3,4,5,6,7,9,11,13,16] & , Fs_{SBR} = 24000 \\ [-6,-4,-2,-1,0,1,2,3,4,5,6,7,9,11,13,16] & , Fs_{SBR} = 32000 \\ [-4,-2,-1,0,1,2,3,4,5,6,7,9,11,13,16,20] & , 44100 \le Fs_{SBR} \le 64000 \\ [-2,-1,0,1,2,3,4,5,6,7,9,11,13,16,20,24] & , Fs_{SBR} > 64000 \end{cases}$$

$$startMin = \begin{cases} NINT\left(3000 \cdot \frac{128}{Fs_{SBR}}\right) & , Fs_{SBR} < 32000 \\ NINT\left(4000 \cdot \frac{128}{Fs_{SBR}}\right) & , 32000 \le Fs_{SBR} < 64000 \\ NINT\left(5000 \cdot \frac{128}{Fs_{SBR}}\right) & , 64000 \le Fs_{SBR} \end{cases}$$

The upper frequency boundary, denoted k_2, is defined by:

$$k_2 = \begin{cases} \min\left(64, stopMin + \sum_{i=0}^{bs\_stop\_freq-1} \mathbf{stopDkSort}\left(i\right)\right) & , 0 \le bs\_stop\_freq < 14 \\ \min\left(64, 2 \cdot k_0\right) & , bs\_stop\_freq = 14 \\ \min\left(64, 3 \cdot k_0\right) & , bs\_stop\_freq = 15 \end{cases}$$

where

$$stopMin = \begin{cases} NINT\left(6000 \cdot \frac{128}{Fs_{SBR}}\right) & , Fs_{SBR} < 32000 \\ NINT\left(8000 \cdot \frac{128}{Fs_{SBR}}\right) & , 32000 \le Fs_{SBR} < 64000 \\ NINT\left(10000 \cdot \frac{128}{Fs_{SBR}}\right) & , 64000 \le Fs_{SBR} \end{cases}$$

$$\mathbf{stopDkSort} = sort\left(\mathbf{stopDk}\right)$$

$$\mathbf{stopDk}\left(p\right) = NINT\left(stopMin \cdot \left(\frac{64}{stopMin}\right)^{\frac{p+1}{13}}\right) - NINT\left(stopMin \cdot \left(\frac{64}{stopMin}\right)^{\frac{p}{13}}\right), 0 \le p \le 12$$

The master frequency band table, $\mathbf{f}_{Master}$, is calculated according to the flowcharts in Figure 4.38 and Figure 4.39. The input variables to the flowcharts are k_0 *and* k_2, as calculated above, and the data elements *bs_freq_scale* and *bs_alter_scale.* The table $\mathbf{f}_{Master}$ is only defined for $k_2 > k_0$. Furthermore, the restrictions in subclause 4.6.18.3.6 apply.

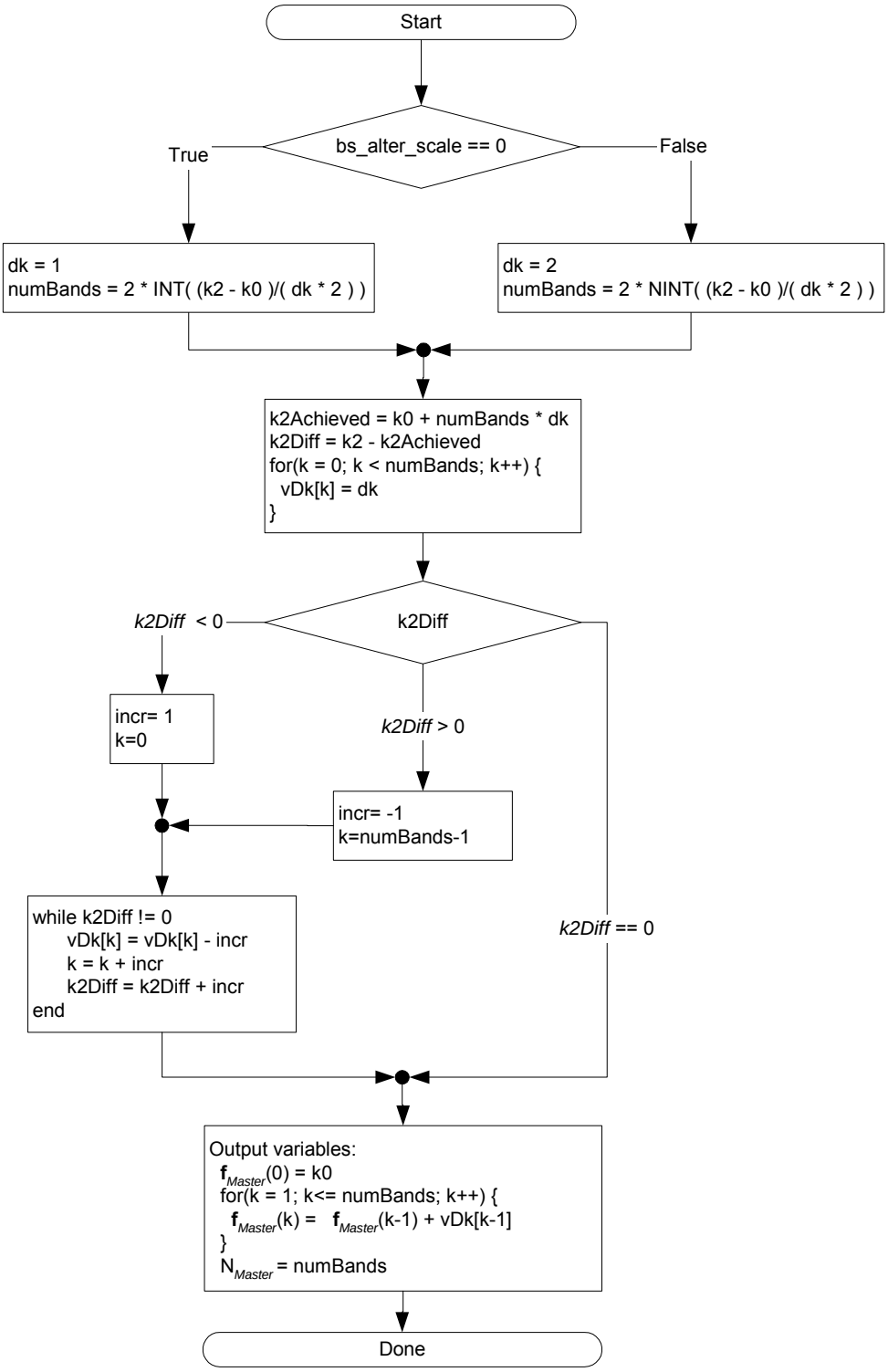


Figure 4.38 – Flowchart calculation of $\mathbf{f}_{Master}$ when *bs_freq_scale = 0*

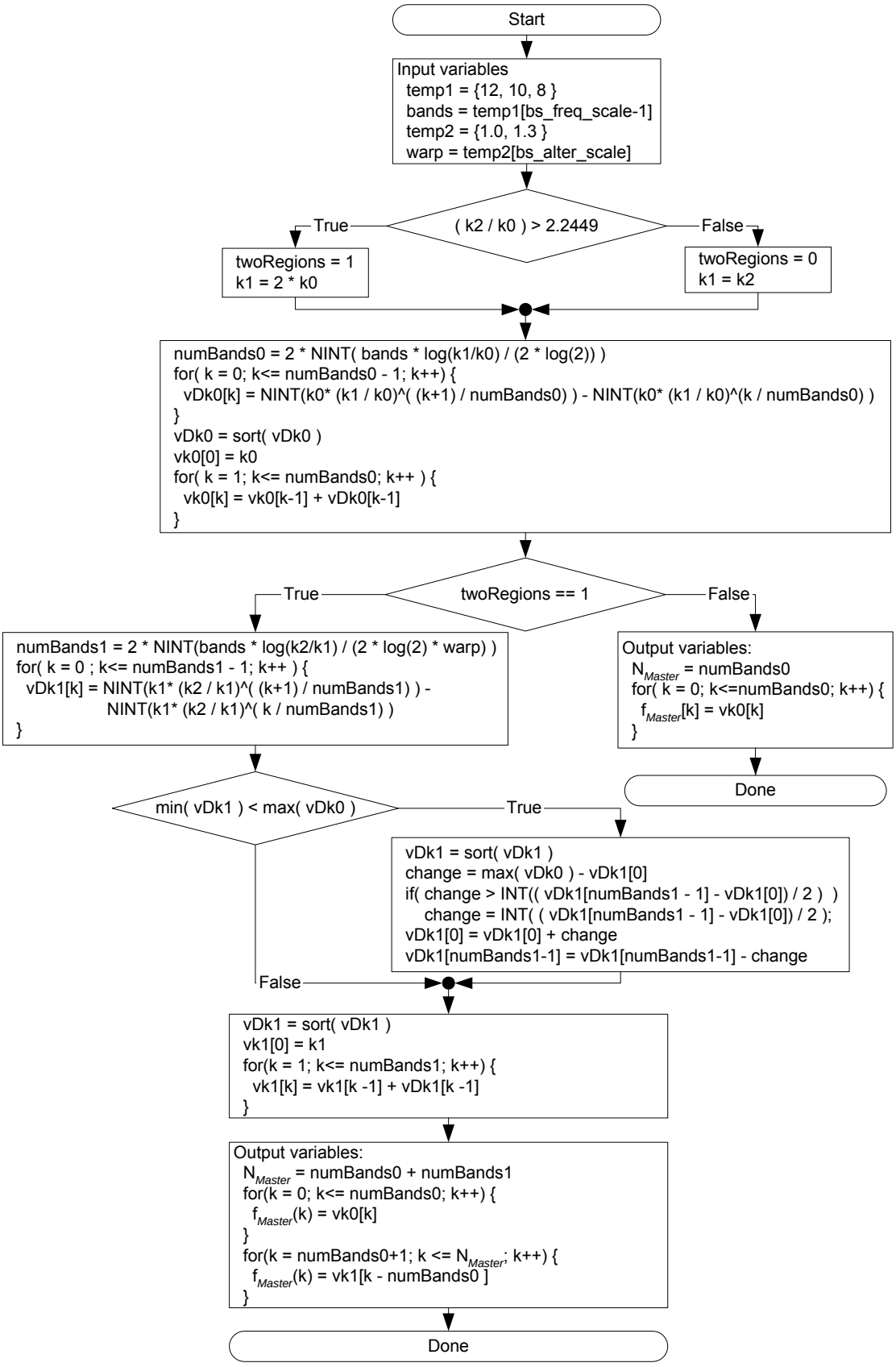


Figure 4.39 – Flowchart calculation of f_{Master} when *bs_freq_scale* > *0*

4.6.18.3.2.2 Derived frequency band tables

The frequency band table $\mathbf{f}_{TableHigh}$, used for high frequency resolution SBR envelopes, is obtained by extracting a subset of the borders from $\mathbf{f}_{Master}$ according to:

$$N_{High} = N_{Master} - bs\_xover\_band$$

$$N_{Low} = INT\left(\frac{N_{High}}{2}\right) + \left(N_{High} - 2 \cdot INT\left(\frac{N_{High}}{2}\right)\right)$$

$$\mathbf{n} = [N_{Low}, N_{High}]$$

$$\mathbf{f}_{TableHigh}(k) = \mathbf{f}_{Master}(k + bs\_xover\_band) \quad , 0 \le k \le N_{High}$$

$$M = \mathbf{f}_{TableHigh}(N_{high}) - \mathbf{f}_{TableHigh}(0)$$

$$k_x = \mathbf{f}_{TableHigh}(0)$$

The frequency band table $\mathbf{f}_{TableLow}$, used for low frequency resolution SBR envelopes, is obtained by extracting a subset of the borders from $\mathbf{f}_{TableHigh}$ according to:

$$\mathbf{f}_{TableLow}(k) = \mathbf{f}_{TableHigh}(i(k)) \quad , 0 \le k \le N_{Low}$$

where $i(k)$ is defined by

$$i(k) = \begin{cases} 0 & if\ k = 0 \\ 2 \cdot k - \dfrac{1 - (-1)^{N_{High}}}{2} & if\ k \neq 0 \end{cases}.$$

The noise floor frequency band table $\mathbf{f}_{TableNoise}$, is extracted from $\mathbf{f}_{TableLow}$ according to

$$\mathbf{f}_{TableNoise}(k) = \mathbf{f}_{TableLow}(i(k)) \quad , 0 \le k \le N_Q$$

where $i(k)$ and N_Q are defined by

$$i(k) = \begin{cases} 0 & if\ k = 0 \\ i(k-1) + INT\left(\dfrac{N_{Low} - i(k-1)}{N_Q + 1 - k}\right) & if\ k \neq 0 \end{cases},$$

$$N_Q = \max\left(1, NINT\left(bs\_noise\_bands \cdot \frac{\log\left(\frac{k_2}{k_x}\right)}{\log(2)}\right)\right) \quad , 0 \le bs\_noise\_bands \le 3 \cdot$$

4.6.18.3.2.3 Limiter frequency band table

The limiter frequency band table $\mathbf{f}_{TableLim}$, is constructed to have either exactly one limiter band over the entire SBR range, or approximately 1.2, 2 or 3 bands per octave, signaled by *bs_limiter_bands* from the bitstream payload. The table holds indices of the synthesis filterbank subbands, where the number of elements equals the number of

bands plus one. The first element is always k_x, $\mathbf{f}_{TableLim}$ is a subset of the union of $\mathbf{f}_{TableLow}$ and the patch borders defined in subclause 4.6.18.6.

If *bs_limiter_bands* is zero only one limiter band is used and $\mathbf{f}_{TableLim}$ is created as

$$\mathbf{f}_{TableLim} = \left[\mathbf{f}_{TableLow}(0), \mathbf{f}_{TableLow}(N_{Low})\right]$$
$$N_L = 1$$

If *bs_limiter_bands* > *0* the limiter frequency resolution table is created according to the flowchart of Figure 4.40. The variables *numPatches*, **patchBorders** and **patchNumSubbands** are calculated in subclause 4.6.18.6.

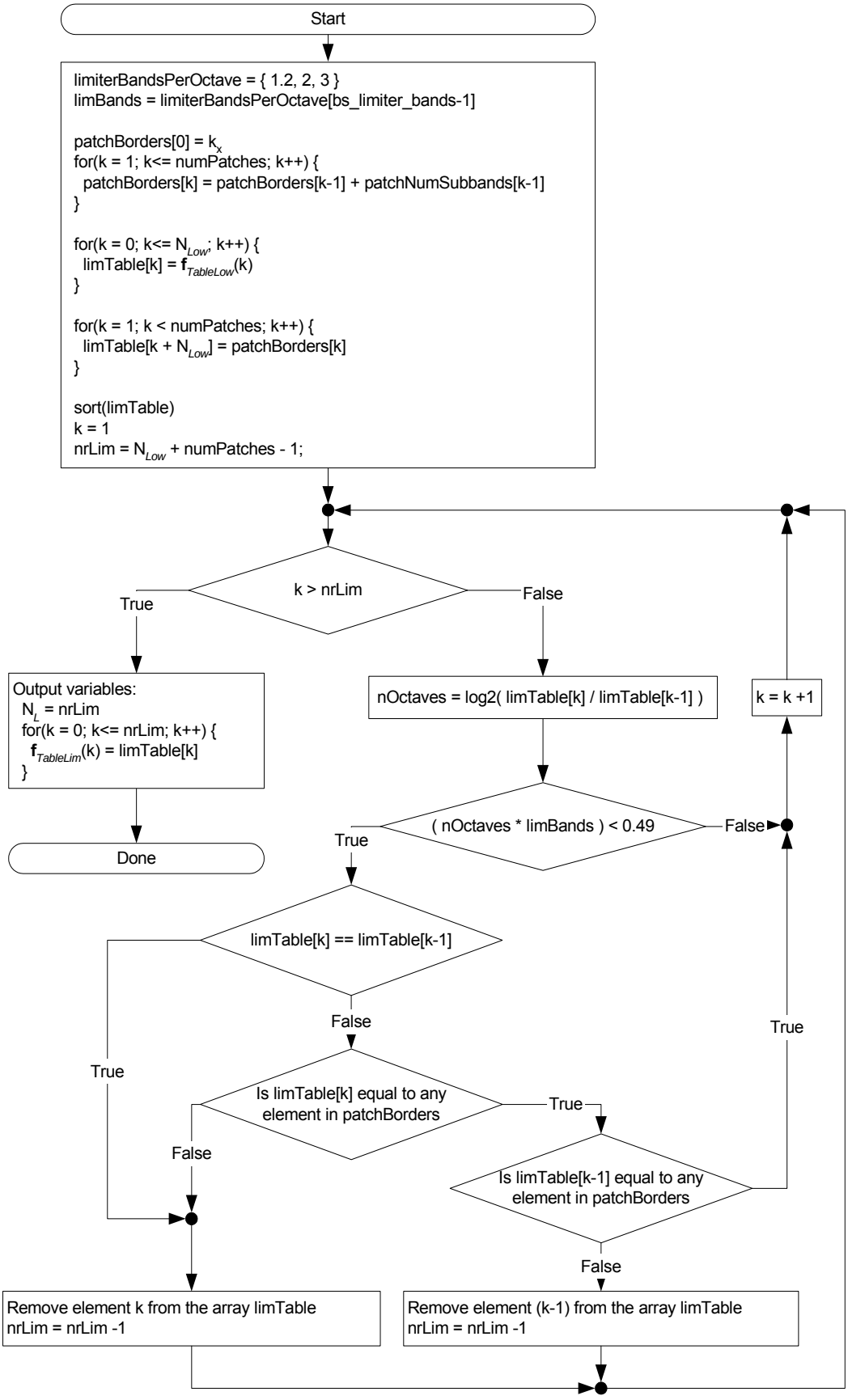


Figure 4.40 – Calculation of $f_{TableLim}$ if *bs_limiter_bands* > *0*

4.6.18.3.3 Time / frequency grid

The time/frequency grid part of the bitstream payload describes the number of SBR envelopes and noise floors as well as the time segment associated with each SBR envelope and noise floor. Furthermore, it describes what frequency band table to use for each SBR envelope. Four different SBR frame classes, FIXFIX, FIXVAR, VARFIX and VARVAR, are used, each of which has different capabilities with respect to time/frequency grid selection. The names refer to whether the locations of the leading and trailing SBR frame borders (i.e. the SBR frame boundaries) are variable or not from a syntactical point of view. The SBR envelope and noise floor time segments are described by the vectors $\mathbf{t}_E(l)$ and $\mathbf{t}_Q(l)$ respectively, which contain the borders for each time segment expressed in time slots. Each time segment is defined by a start time border and a stop time border. The time slot indicated by the start time border is included in the time segment, the time slot indicated by the stop time border is excluded from the time segment. For both vectors, the stop time border of time segment l equals the start time border of time segment $l+1$, where l is an arbitrary time segment in the vector. The calculation of $\mathbf{t}_E(l)$ is described below.

First the leading SBR frame border, *absBordLead*, and the trailing SBR frame border, *absBordTrail*, are obtained from the bitstream payload according to:

$$absBordLead = \begin{cases} 0 & ,bs\_frame\_class = FIXFIX \ or \ FIXVAR \\ bs\_var\_bord\_0 & ,bs\_frame\_class = VARVAR \ or \ VARFIX \end{cases}$$

$$absBordTrail = \begin{cases} numTimeSlots & ,bs\_frame\_class = FIXFIX \ or \ VARFIX \\ bs\_var\_bord\_1 + numTimeSlots & ,bs\_frame\_class = VARVAR \ or \ FIXVAR \end{cases}.$$

In order to decode the time borders of all SBR envelopes within the SBR frame, the number of relative borders associated with the leading and trailing time borders respectively are calculated according to:

$$n_{RelLead} = \begin{cases} L_E - 1 & ,bs\_frame\_class = FIXFIX \\ 0 & ,bs\_frame\_class = FIXVAR \\ bs\_num\_rel\_0 & ,bs\_frame\_class = VARVAR \ or \ VARFIX \end{cases}$$

$$n_{RelTrail} = \begin{cases} 0 & ,bs\_frame\_class = FIXFIX \ or \ VARFIX \\ bs\_num\_rel\_1 & ,bs\_frame\_class = VARVAR \ or \ FIXVAR \end{cases}$$

where

$$L_E = bs\_num\_env$$

The SBR envelope time border vector of the current SBR frame, $\mathbf{t}_E(l)$, is calculated according to:

$$\mathbf{t}_E(l) = \begin{cases} absBordLead & if \ \ l = 0 \\ absBordTrail & if \ \ l = L_E \\ absBordLead + \sum_{i=0}^{l-1} \mathbf{relBordLead}(i) & if \ \ 1 \le l \le n_{RelLead} \\ absBordTrail - \sum_{i=0}^{L_E - l - 1} \mathbf{relBordTrail}(i) & if \ \ n_{RelLead} < l < L_E \end{cases}$$

where $0 \le l \le L_E$ and $\mathbf{relBordLead}(l)$ and $\mathbf{relBordTrail}(l)$ are vectors containing the relative borders associated with the leading and trailing borders respectively. Both vectors are (if applicable) defined below.

$$\mathbf{relBordLead}(l) = \begin{cases} NINT\left(\dfrac{numTimeSlots}{L_E}\right) & ,bs\_frame\_class = FIXFIX \\ NA & ,bs\_frame\_class = FIXVAR \\ \mathbf{bs\_rel\_bord\_0}(l) & ,bs\_frame\_class = VARVAR\ or\ VARFIX \end{cases}$$

where $0 \le l < n_{RelLead}$

$$\mathbf{relBordTrail}(l) = \begin{cases} NA & ,bs\_frame\_class = FIXFIX\ or\ VARFIX \\ \mathbf{bs\_rel\_bord\_1}(l) & ,bs\_frame\_class = VARVAR\ or\ FIXVAR \end{cases}$$

where $0 \le l < n_{RelTrail}$

Within one SBR frame there can either be one or two noise floors. The noise floor time borders are derived from the SBR envelope time border vector according to:

$$L_Q = bs\_num\_noise$$

$$\mathbf{t}_Q = \begin{cases} \left[\mathbf{t}_E(0), \mathbf{t}_E(1)\right] & ,L_E = 1 \\ \left[\mathbf{t}_E(0), \mathbf{t}_E(middleBorder), \mathbf{t}_E(L_E)\right] & ,L_E > 1 \end{cases}$$

where *middleBorder* = *func*(*bs_frame_class*, *bs_pointer*, L_E) is calculated according to Table 4.155 below.

Table 4.155 – *middleBorder* function

bs_pointer	*bs_frame_class*		
	FIXFIX	*VARFIX*	*FIXVAR,VARVAR*
=0	$\frac{L_E}{2}$	1	$L_E - 1$
=1	$\frac{L_E}{2}$	$L_E - 1$	$L_E - 1$
>1	$\frac{L_E}{2}$	$bs\_pointer - 1$	$L_E + 1 - bs\_pointer$

As previously stated, each SBR envelope can be of either high or low frequency resolution. This is described by an SBR envelope frequency resolution vector, **r**(l), which is calculated according to:

$$\mathbf{r}(l) = \mathbf{bs\_freq\_res}(l) \quad ,0 \le l < L_E$$

where

$\mathbf{r}(l) = 0$ signals the usage of $\mathbf{f}_{TableLow}$ for SBR envelope l

$\mathbf{r}(l) = 1$ signals the usage of $\mathbf{f}_{TableHigh}$ for SBR envelope l

4.6.18.3.4 SBR envelope and noise floor decoding

Delta coding of envelope scalefactors and noise floor scalefactors is done in either time or frequency direction for each SBR envelope, and noise floor. When delta coding in the time direction across SBR frame boundaries is applied, the first SBR envelope in the current SBR frame is delta coded with respect to the last SBR envelope of the previous SBR frame. The same is true for the noise floors.

If the SBR Tool is used with a scalable AAC codec, the envelope scalefactors of a stereo enhancement layer can only be decoded if the enhancement layer is available and the variable *enhanceLayDecE* is one. The *enhanceLayDecE* variable is defined as:

$$enhanceLayDecE = \begin{cases} 1 & , if\ enhanceLayDecE' = 1 \quad or \quad bs\_df\_env(0) = 0 \\ 0 & , otherwise \end{cases}$$

where *enhanceLayDecE'* represents the value of the previous SBR frame. For the first SBR frame *enhanceLayDecE'* is set to one. If a scalable system is not used, the variable *enhanceLayDecE* shall be a constant set to one.

The deduction of the envelope scalefactors $\mathbf{E}$ from the delta coded envelope scalefactors $\mathbf{E}_{Delta}$ is defined by:

$$\mathbf{E}(k,l) = \begin{cases} \sum_{i=0}^{k} \delta \cdot \mathbf{E}_{Delta}(i,l) & , \begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \\ \mathbf{bs\_df\_env}(l) = 0 \end{cases} \\ g_E(k,l) + \delta \cdot \mathbf{E}_{Delta}(k,l) & , \begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \\ \mathbf{bs\_df\_env}(l) = 1 \\ \mathbf{r}(l) = g(l) \end{cases} \\ g_E(i(k),l) + \delta \cdot \mathbf{E}_{Delta}(k,l) & , \begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \\ \mathbf{bs\_df\_env}(l) = 1 \\ \mathbf{r}(l) = 0 \\ g(l) = 1 \\ i(k) \text{ is defined by} \\ \mathbf{f}_{TableHigh}(i(k)) = \mathbf{f}_{TableLow}(k) \end{cases} \\ g_E(i(k),l) + \delta \cdot \mathbf{E}_{Delta}(k,l) & , \begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \\ \mathbf{bs\_df\_env}(l) = 1 \\ \mathbf{r}(l) = 1 \\ g(l) = 0 \\ i(k) \text{ is defined by} \\ \mathbf{f}_{TableLow}(i(k)) \le \mathbf{f}_{TableHigh}(k) < \mathbf{f}_{TableLow}(i(k)+1) \end{cases} \end{cases}$$

where

$$\delta = \begin{cases} 2 & if\ ch = 1\ AND\ bs\_coupling = 1 \\ 1 & otherwise \end{cases}$$

and where $g_E(k,l)$ and $g(l)$ are defined below and $\mathbf{E}_{Delta}(k,l)$ is read from the data element *bs_data_env* as shown below. The envelope scalefactors from the previous SBR frame, $\mathbf{E}'$, are needed when delta coding in the time direction over SBR frame boundaries. The number of SBR envelopes of the previous SBR frame is denoted L'_E, and is also needed in that case, as well as the frequency resolution vector of the previous SBR frame, denoted $\mathbf{r}'$.

$$g_E(k,l) = \begin{cases} \mathbf{E}(k,l-1) & , \begin{cases} 1 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \end{cases} \\ \mathbf{E}'(k, L'_E - 1) & , \begin{cases} l = 0 \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \end{cases} \end{cases} \quad \text{and} \quad g(l) = \begin{cases} \mathbf{r}(l-1) & , 1 \le l < L_E \\ \mathbf{r}'(L'_E - 1) & , l = 0 \end{cases}.$$

$$\mathbf{E}_{Delta}(k,l) = bs\_data\_env[ch][l][k] \quad , \begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \end{cases}$$

If the SBR Tool is used with a scalable AAC codec, the noise floor data of a stereo enhancement layer can only be decoded if the enhancement layer is available and the variable *enhanceLayDecQ* is one. The *enhanceLayDecQ* variable is defined as:

$$enhanceLayDecQ = \begin{cases} 1 & , if\ enhanceLayDecQ' = 1 \quad or \quad bs\_df\_noise(0) = 0 \\ 0 & , otherwise \end{cases}$$

where *enhanceLayDecQ'* represents the value of the previous SBR frame. For the first SBR frame *enhanceLayDecQ'* is set to one. If a scalable system is not used, the variable *enhanceLayDecQ* shall be a constant set to one.

The deduction of the noise floor data $\mathbf{Q}$ from the delta coded noise floor data $\mathbf{Q}_{Delta}$ is defined by.

$$\mathbf{Q}(k,l) = \begin{cases} \sum_{i=0}^{k} \delta \cdot \mathbf{Q}_{Delta}(i,l) & , \begin{cases} 0 \le l < L_Q \\ 0 \le k < N_Q \\ \mathbf{bs\_df\_noise}(l) = 0 \end{cases} \\ \mathbf{Q}(k,l-1) + \delta \cdot \mathbf{Q}_{Delta}(k,l) & , \begin{cases} 1 \le l < L_Q \\ 0 \le k < N_Q \\ \mathbf{bs\_df\_noise}(l) = 1 \end{cases} \\ \mathbf{Q}'(k, L'_Q - 1) + \delta \cdot \mathbf{Q}_{Delta}(k,0) & , \begin{cases} l = 0 \\ 0 \le k < N_Q \\ \mathbf{bs\_df\_noise}(0) = 1 \end{cases} \end{cases}$$

where

$$\delta = \begin{cases} 2 & if\ ch = 1\ AND\ bs\_coupling = 1 \\ 1 & otherwise \end{cases}$$

and where $\mathbf{Q}'$ is the noise floor scalefactors from the previous SBR frame and L'_Q is the number of noise floors from the previous SBR frame. $\mathbf{Q}_{Delta}(k,l)$ is read from the data element *bs_data_noise* as shown below.

$$\mathbf{Q}_{Delta}(k,l) = bs\_data\_noise[ch][l][k] \quad ,\begin{cases} 0 \le l < L_Q \\ 0 \le k < N_Q \end{cases}$$

4.6.18.3.5 Dequantization and stereo decoding

For the quantization of the envelope scalefactors, there are two quantization steps available. *bs_amp_res* = 0 corresponds to a quantization step of 1.5 dB and *bs_amp_res* = 1 corresponds to a quantization step of 3.0 dB.

For a single channel element, the envelope scalefactors are dequantized according to:

$$\mathbf{E}_{Orig}(k,l) = 64 \cdot 2^{\frac{\mathbf{E}(k,l)}{a}} \quad ,\begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \end{cases}$$

where $a = \begin{cases} 2 & ,bs\_amp\_res = 0 \\ 1 & ,bs\_amp\_res = 1 \end{cases}$.

The noise floor scalefactors are dequantized according to:

$$\mathbf{Q}_{Orig}(k,l) = 2^{NOISE\_FLOOR\_OFFSET - \mathbf{Q}(k,l)} \quad ,\begin{cases} 0 \le l < L_Q \\ 0 \le k < N_Q \end{cases}$$

For a channel pair element where coupling mode is not used, i.e. *bs_coupling* = 0, the individual channels are treated as the single channel element case above.

If coupling mode is used, i.e. *bs_coupling* = 1, the time-grids $\mathbf{t}_E$ and $\mathbf{t}_Q$ are the same for both channels. Let $\mathbf{E}_0$, $\mathbf{E}_1$, $\mathbf{Q}_0$, and $\mathbf{Q}_1$ represent the decoded envelope scalefactors and noise floor scalefactors, in accordance with the decoding process outlined above. Subscript zero represents the firstly decoded channel (the energy average and the noise-floor average of the original left and right channel) and subscript one represents the secondly decoded channel (the energy ratio and the noise-floor ratio of the original left and right channel).

Below it is shown how to dequantize the envelope scalefactors and noise floor scalefactors in coupling mode (*bs_coupling* = 1).

$$\mathbf{E}_{LeftOrig}(k,l) = 64 \cdot \frac{2^{\frac{\mathbf{E}_0(k,l)}{a}+1}}{1+2^{\frac{\mathbf{panOffset}(bs\_amp\_res) - \mathbf{E}_1(k,l)}{a}}} \quad ,\begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \end{cases}$$

$$\mathbf{E}_{RightOrig}(k,l) = 64 \cdot \frac{2^{\frac{\mathbf{E}_0(k,l)}{a}+1}}{1+2^{\frac{\mathbf{E}_1(k,l) - \mathbf{panOffset}(bs\_amp\_res)}{a}}} \quad ,\begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \end{cases}$$

$$\mathbf{Q}_{OrigLeft}(k,l) = \frac{2^{NOISE\_FLOOR\_OFFSET - \mathbf{Q}_0(k,l)+1}}{1+2^{\mathbf{panOffset}(1) - \mathbf{Q}_1(k,l)}} \quad ,\begin{cases} 0 \le l < L_Q \\ 0 \le k < N_Q \end{cases}$$

$$\mathbf{Q}_{OrigRight}(k,l) = \frac{2^{NOISE\_FLOOR\_OFFSET - \mathbf{Q}_0(k,l)+1}}{1+2^{\mathbf{Q}_1(k,l) - \mathbf{panOffset}(1)}} \quad ,\begin{cases} 0 \le l < L_Q \\ 0 \le k < N_Q \end{cases}.$$

Above, the SBR envelopes and noise floors are denoted $\mathbf{E}_{OrigLeft}$, $\mathbf{E}_{OrigRight}$, $\mathbf{Q}_{OrigLeft}$ and $\mathbf{Q}_{OrigRight}$, in order to differentiate between the two channels in the channel pair element. Since no channel dependency exists after the above decoding and dequantization, the envelopes and noise floors will from this point on be referred to as $\mathbf{E}_{Orig}$ and $\mathbf{Q}_{Orig}$.

If the SBR Tool is used with a scalable AAC codec and a stereo enhancement layer is not present, the data in the channel pair element available in the mono base layer shall be dequantized as a single channel element, i.e. $\mathbf{E}_{LeftOrig}(k,l)$ is calculated as $\mathbf{E}_{Orig}(k,l)$.

If the stereo enhancement layer is available, the envelope scalefactors in the channel pair element shall be dequantized and stereo decoded as a channel pair element, provided that the variable *enhanceLayDecE* is one. If the variable *enhanceLayDecE* is zero, the envelope scalefactors of the first channel are dequantized as a single channel element, i.e. $\mathbf{E}_{LeftOrig}(k,l)$ is calculated as $\mathbf{E}_{Orig}(k,l)$, and the envelope scalefactors of the second channel, shall be set to the same values as the envelope scalefactors of the first channel, i.e. $\mathbf{E}_{RightOrig}(k,l) = \mathbf{E}_{LeftOrig}(k,l)$.

The same applies for the noise floor scalefactors, i.e. if the a stereo enhancement layer is not present, the data in the channel pair element available in the mono base layer shall be dequantized as a single channel element, i.e. $\mathbf{Q}_{LeftOrig}(k,l)$ is calculated as $\mathbf{Q}_{Orig}(k,l)$.

If the stereo enhancement layer is available, the noise floor scalefactors in the channel pair element shall be dequantized and stereo decoded as a channel pair element, provided that the variable *enhanceLayDecQ* is one. If the variable *enhanceLayDecQ* is zero, the noise floor scalefactors of the first channel are dequantized as a single channel element, i.e. $\mathbf{Q}_{LeftOrig}(k,l)$ is calculated as $\mathbf{Q}_{Orig}(k,l)$, and the noise floor scalefactors of the second channel, shall be set to the same values as the noise floor scalefactors of the first channel, i.e. $\mathbf{Q}_{RightOrig}(k,l) = \mathbf{Q}_{LeftOrig}(k,l)$.

4.6.18.3.6 Requirements

The following requirements apply to the SBR data.

- The number of QMF subbands covered by SBR, i.e. k_2-k_0, shall satisfy:
$$k_2 - k_0 \le \begin{cases} 48 & , Fs_{SBR} \le 32kHz \\ 35 & , Fs_{SBR} = 44.1kHz \\ 32 & , Fs_{SBR} \ge 48kHz \end{cases}.$$
- The stop frequency border of the SBR range shall be within $\frac{Fs_{SBR}}{2}$, i.e. $k_x + M \le 64$.
- The start frequency border of the SBR range shall be within $\frac{Fs_{SBR}}{4}$, i.e. $k_x \le 32$.
- The number of SBR envelopes in an SBR frame, L_E, shall satisfy:
$$L_E \le \begin{cases} 4 & , bs\_frame\_class = FIXFIX \\ 5 & , bs\_frame\_class = VARVAR \end{cases}$$
- The number of noise floor scale factors, N_Q, shall satisfy $N_Q \le 5$.
- The number of patches *numPatches*, shall satisfy $numPatches \le 5$.
- For single channel elements and for channel pair elements where coupling is not used (i.e. *bs_coupling* = 0), the quantized noise floor scalefactors $\mathbf{Q}$ shall satisfy: $\mathbf{Q} \in [0,30]$.
- For channel pair elements where coupling is used (i.e. *bs_coupling* = 1), the quantized noise floor scalefactors shall satisfy:
  - $\mathbf{Q}_0 \in [0,30]$
  - $\mathbf{Q}_1 \in [0, 2 \cdot \mathbf{panOffset}(1)]$
- For channel pair elements where coupling is used (i.e. *bs_coupling* = 1), the quantized noise floor scalefactors for the second decoded channel (i.e. $\mathbf{Q}_1$), and the quantized envelope scalefactors for the second decoded channel (i.e. $\mathbf{E}_1$) shall be an even integer.
- In the flowchart in Figure 4.39, when the flowchart has been executed, the following relations shall be satisfied:
  - $numBands0 > 0$
  - $vDk0(i) > 0 \; \forall \; i$

- $vDk1(i) > 0 \; \forall \; i$

- In the flowchart in Figure 4.38, when the flowchart has been executed, the variable *numBands* shall satisfy: $numBands > 0$.
- The following relation shall be satisfied: $bs\_xover\_band < N_{Master}$.
- Delta coded envelope scalefactors and noise floor scalefactors, shall be within the range of the Huffman tables in 4.A.6.1
- If the SBR tool is used in a system that operates with mono/stereo scalability, the *bs_coupling* bit shall be set to one.
- If the SBR tool is used in a system that operates with bandwidth or mono/stereo scalability, all SBR bitstream payload, except the stereo enhancement part of the SBR data, shall be placed in the lowest layer of the data stream. The stereo enhancement part of the SBR data that shall be placed in the lowest layer of the data stream carrying stereo data. All SBR data shall cover the largest SBR range that can occur for the different layers in the stream.

4.6.18.4 SBR filterbanks

4.6.18.4.1 Analysis filterbank

A QMF bank is used to split the time domain signal output from the core decoder into 32 subband signals. The output from the filterbank, i.e. the subband samples, are complex-valued and thus oversampled by a factor two compared to a regular QMF bank. The flowchart of this operation is given in Figure 4.41. The filtering involves the following steps, where an array $\mathbf{x}$ consisting of 320 time domain input samples is assumed. A higher index into the array corresponds to older samples.

- Shift the samples in the array $\mathbf{x}$ by 32 positions. The oldest 32 samples are discarded and 32 new samples are stored in positions 0 to 31.
- Multiply the samples of array $\mathbf{x}$ by every other coefficient of window $\mathbf{c}$. The window coefficients can be found in Table 4.A.87.
- Sum the samples according to the formula in the flowchart to create the 64-element array $\mathbf{u}$.
- Calculate 32 new subband samples by the matrix operation $\mathbf{Mu}$, where $\mathbf{M}(k,n) = 2 \cdot \exp\left(\frac{i \cdot \pi \cdot (k + 0.5) \cdot (2 \cdot n - 0.5)}{64}\right) \; , \begin{cases} 0 \le k < 32 \\ 0 \le n < 64 \end{cases}$

  In the equation, $\exp()$ denotes the complex exponential function and i is the imaginary unit.

Every loop in the flowchart produces 32 complex-valued subband samples, each representing the output from one filterbank subband. For every SBR frame the filterbank will produce $numTimeSlots \cdot RATE$ subband samples for every subband, corresponding to a time domain signal of length $numTimeSlots \cdot RATE \cdot 32$ samples. In the flowchart $\mathbf{W}$[k][l] corresponds to subband sample l in QMF subband k.

4.6.18.4.2 Synthesis filterbank

Synthesis filtering of the SBR-processed subband signals is achieved using a 64-subband QMF bank. The output from the filterbank is real-valued time domain samples. The process is given by the flowchart inFigure 4.42. The synthesis filtering comprises the following steps, where an array $\mathbf{v}$ consisting of 1280 samples is assumed:

- Shift the samples in the array $\mathbf{v}$ by 128 positions. The oldest 128 samples are discarded.
- The 64 new complex-valued subband samples are multiplied by the matrix $\mathbf{N}$, where $\mathbf{N}(k,n) = \frac{1}{64} \cdot \exp\left(\frac{i \cdot \pi \cdot (k + 0.5) \cdot (2 \cdot n - 255)}{128}\right) \; , \begin{cases} 0 \le k < 64 \\ 0 \le n < 128 \end{cases}$

  In the equation, $\exp()$ denotes the complex exponential function and i is the imaginary unit. The real part of the output from this operation is stored in the positions 0 to 127 of array $\mathbf{v}$.

- Extract samples from v according to the flowchart in inFigure 4.42 to create the 640-element array **g**.
- Multiply the samples of array **g** by window **c** to produce array **w**. The window coefficients of **c** can be found in Table 4.A.87, and are the same as for the analysis filterbank.
- Calculate 64 new output samples by summation of samples from array w according to the last step in the flowchart of inFigure 4.42

Every SBR frame produces an output of $numTimeSlots \cdot RATE \cdot 64$ time domain samples. In the flowchart of in Figure 4.42 **X**[k][l] corresponds to subband sample l in the QMF subband k, and every new loop produces 64 time domain samples as output.

4.6.18.4.3 Downsampled synthesis filterbank

The downsampled synthesis filtering of the SBR-processed subband signals is achieved using a 32-channel QMF bank. The output from the filterbank is real-valued time domain samples. The process is given of the flowchart in Figure 4.43. The synthesis filtering comprises the following steps, where an array **v** consisting of 640 samples is assumed:

- Shift the samples in the array **v** by 64 positions. The oldest 64 samples are discarded.
- The 32 new complex-valued subband samples are multiplied by the matrix **N**, where $\mathbf{N}(k,n) = \frac{1}{64} \cdot \exp\left(\frac{i \cdot \pi \cdot (k+0.5) \cdot (2 \cdot n - 127)}{64}\right)$, $\begin{cases} 0 \le k < 32 \\ 0 \le n < 64 \end{cases}$

  In the equation, exp() denotes the complex exponential function and i is the imaginary unit. The real part of the output from this operation is stored in the positions 0 to 63 of array **v**.
- Extract samples from **v** according to the flowchart in Figure 4.43 to create the 320-element array **g**.
- Multiply the samples of array **g** by every other coefficient of window **w**. The window coefficients of **c** can be found in Table 4.A.87, and are the same as for the analysis filterbank.
- Calculate 32 new output samples by summation of samples from array **w** according to the last step in the flowchart of Figure 4.43

Every SBR frame produces an output of $numTimeSlots \cdot RATE \cdot 32$ time domain samples. In the flowchart of Figure 4.43 **X**[k][l] corresponds to subband sample l in the QMF subband k, and every new loop produces 32 time domain samples as output.

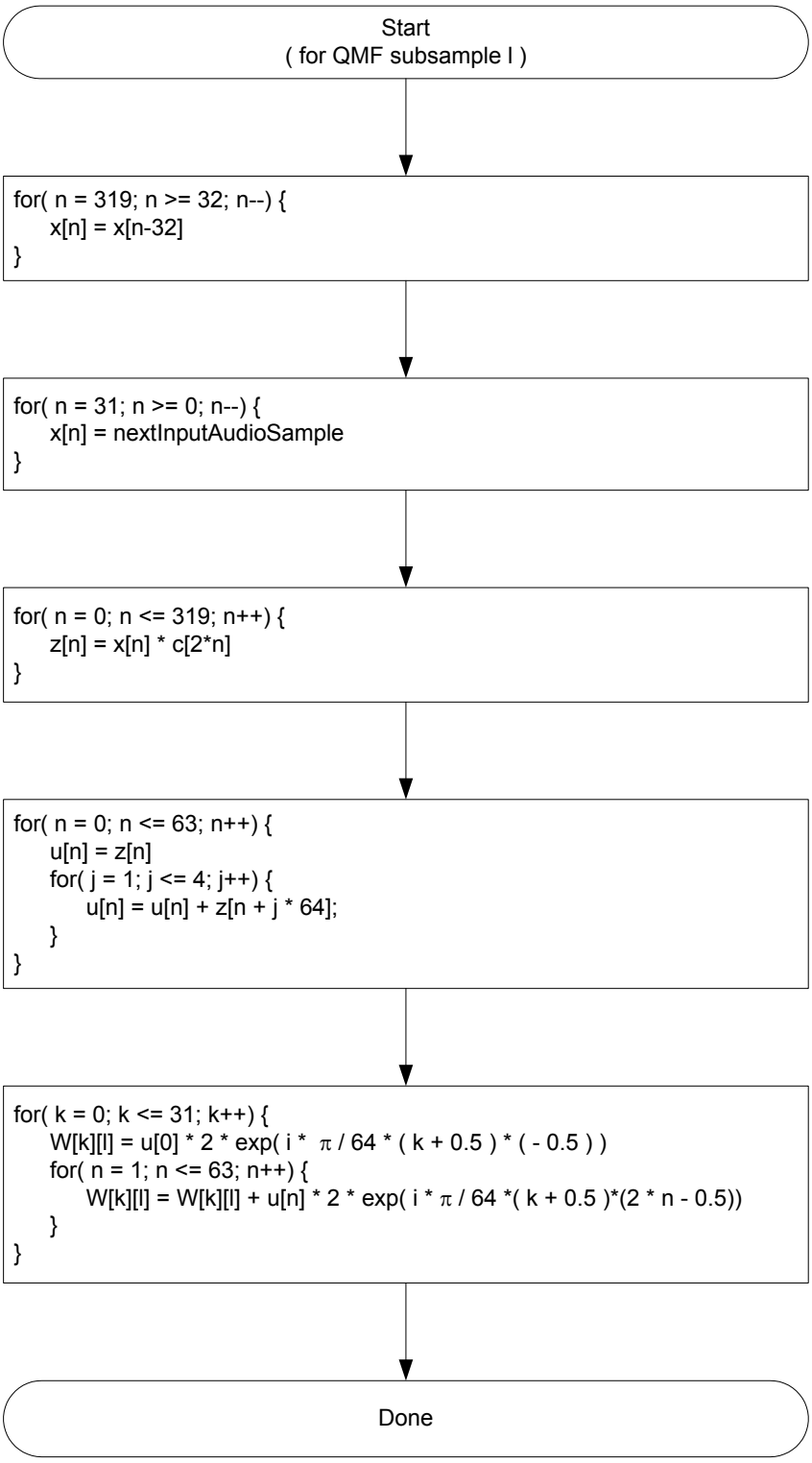


Figure 4.41 – Flowchart of decoder analysis QMF bank

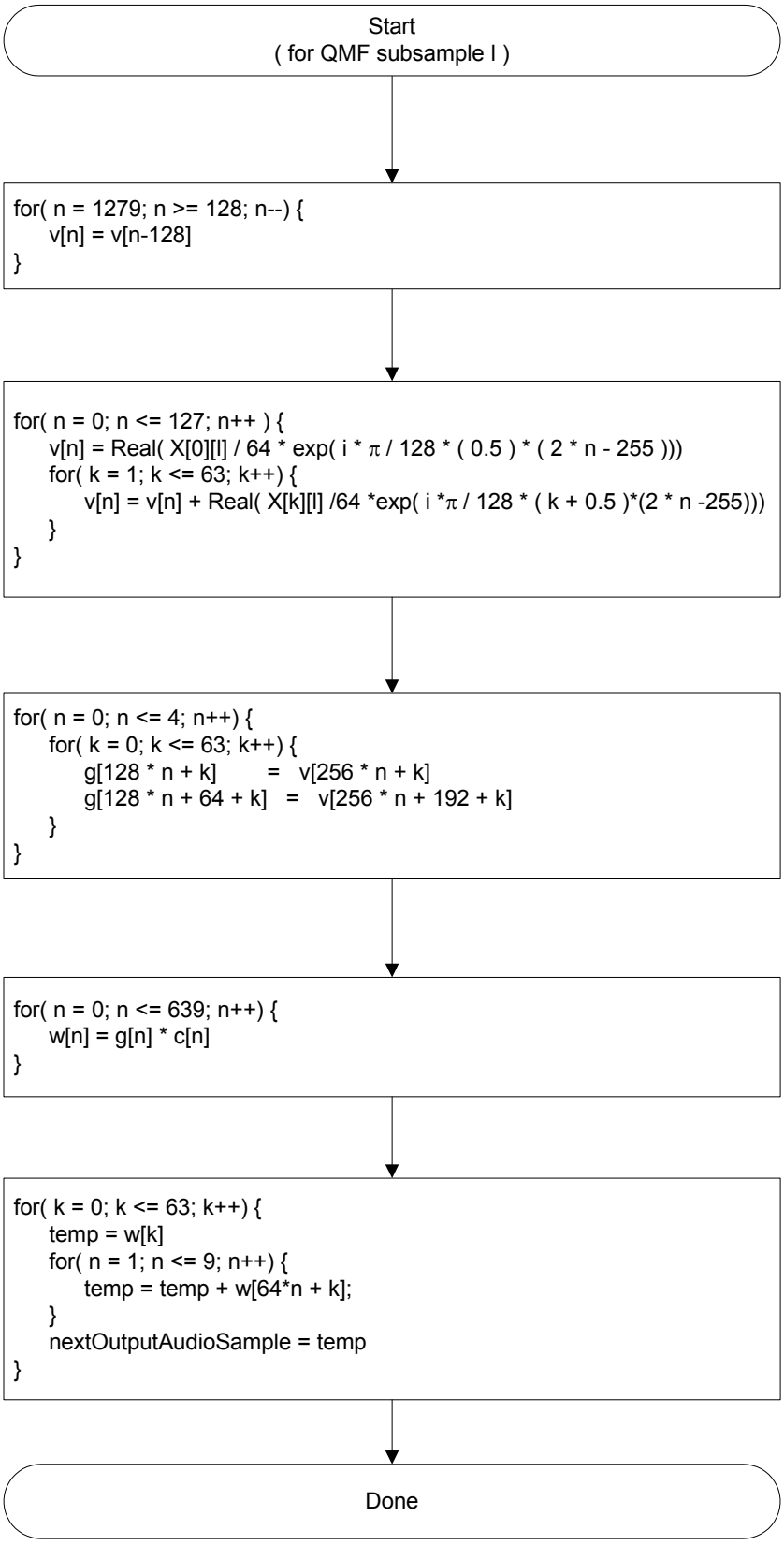


Figure 4.42 – Flowchart of decoder synthesis QMF bank

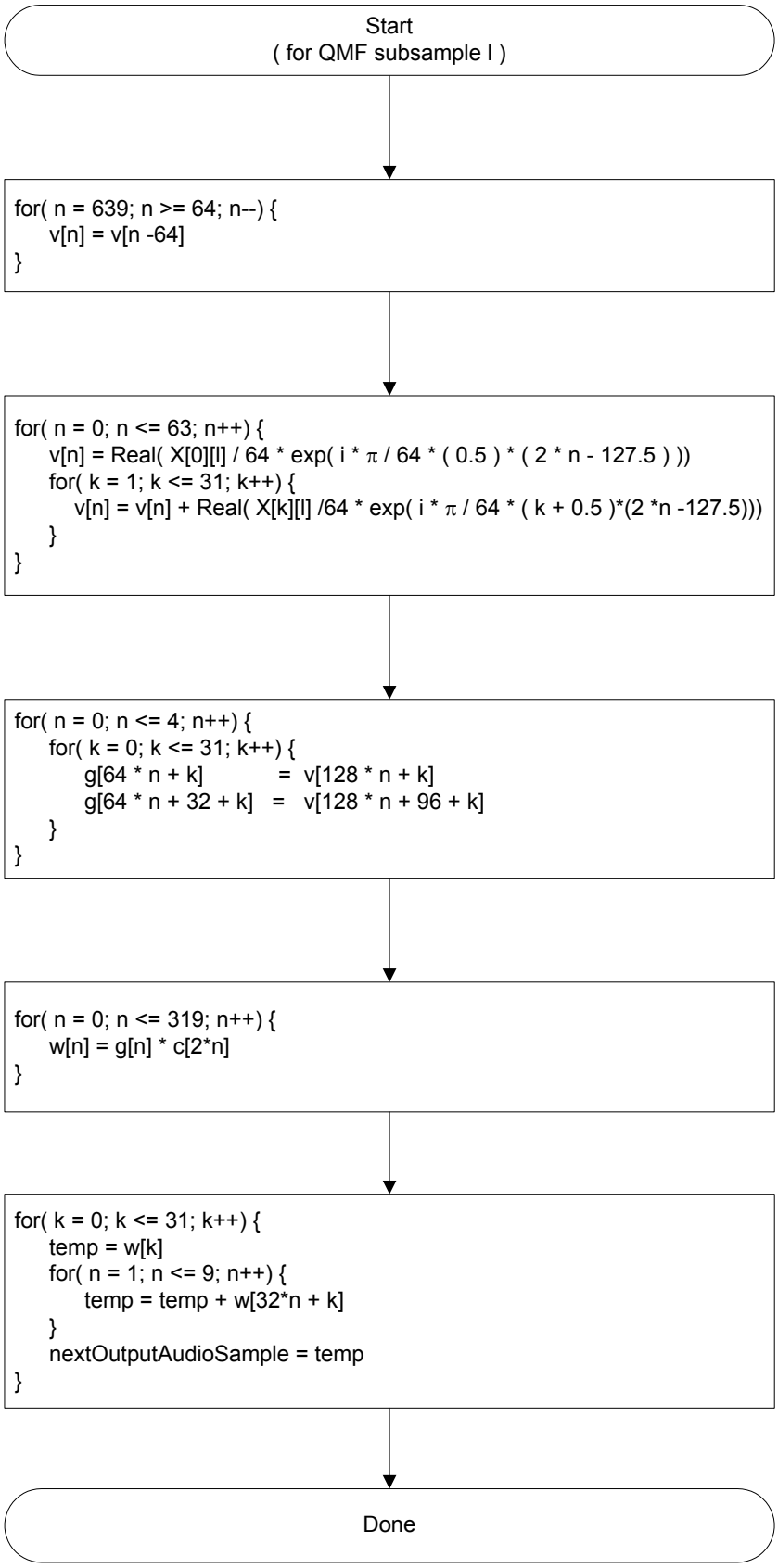


Figure 4.43 – Flowchart of decoder downsampled synthesis QMF bank

4.6.18.5 SBR tool overview

The decoder block diagram of Figure 4.44 shows how the SBR parts and the AAC core decoder are interconnected. In order to synchronize the SBR envelope data and the AAC core decoder output, the SBR bitstream payload has to be time delayed with respect to the AAC core bitstream payload, i.e. the SBR parts in the encoder are operating on time delayed audio samples with respect to the AAC core encoder. To achieve a synchronized output signal, the following steps have to be acknowledged in the decoder:

- The bitstream payload deformatter divides the bitstream payload into two parts; the AAC core coder part and the SBR part.
- The SBR bitstream payload part is fed to the bitstream payload parser followed by dequantization. The raw data is Huffman decoded.
- The AAC bitstream payload part is fed to the AAC core decoder, where the bitstream payload of the current SBR frame is decoded, yielding a time domain audio signal block of 1024 samples or 960 samples depending on frame size.
- The core coder audio block is fed to the analysis QMF bank. This is illustrated in Figure 4.45 (a) by the dashed block. If a scalable core coder is used the audio block representing the highest available layer shall be used.
- The analysis QMF bank performs the filtering of the core coder audio signal. Section 4.6.18.4.1 describes the analysis filter bank and Figure 4.45 (a) shows the timing of the analysis windowing. The subband filtered low band is defined by $\mathbf{X}_{Low}$ according to:

$$\mathbf{X}_{Low}(k,\ l\ ) = \begin{cases} \mathbf{W}(k, l - t_{HFGen}) & ,\ 0 \le k < k_x,\ t_{HFGen} \le l < l_f + t_{HFGen} \\ 0 & ,\ k_x \le k < 32,\ t_{HFGen} \le l < l_f + t_{HFGen} \\ \mathbf{W}'(k, l + l_f - t_{HFGen}) & ,\ 0 \le k < k'_x,\ 0 \le l < t_{HFGen} \\ 0 & ,\ k'_x \le k < 32,\ 0 \le l < t_{HFGen} \end{cases}$$

  where $\mathbf{W}'$ is the $\mathbf{W}$ matrix from the previous frame, and k'_x is the k_x value from the previous frame, and where $l_f = numTimeSlots \cdot RATE$. If scalable SBR is used, or if the SBR tool is used for pure upsampling without SBR processing, the following apply instead of the equation above:

$$\mathbf{X}_{Low}(k,\ l\ ) = \begin{cases} \mathbf{W}(k, l - t_{HFGen}) & ,\ 0 \le k < 32,\ t_{HFGen} \le l < l_f + t_{HFGen} \\ \mathbf{W}'(k, l + l_f - t_{HFGen}) & ,\ 0 \le k < 32,\ 0 \le l < t_{HFGen} \end{cases}$$

- The output from the analysis QMF bank is hence delayed t_{HFGen} subband samples, before being fed to the synthesis QMF bank. To achieve synchronization $t_{HFGen} = 8$. The resulting subband samples are shown in Figure 4.45 (b) as the upper dashed block.
- The HF generator calculates $\mathbf{X}_{High}$ given the matrix $\mathbf{X}_{Low}$ according to the scheme outlined in section 4.6.18.6. The process is guided by the SBR data contained in the current SBR frame. The result is illustrated by the dashed block in Figure 4.45 (b).
- The envelope adjuster outlined in subclause 4.6.18.7 calculates the matrix $\mathbf{Y}$ given the matrix $\mathbf{X}_{High}$ and the SBR envelope data, extracted from the SBR bitstream payload. To achieve synchronization, t_{HFAdj} has to be set to $t_{HFAdj} = 2$, i.e. the envelope adjuster operates on data delayed t_{HFGen} - t_{HFAdj} subband samples.
- The synthesis QMF bank operates on the output from the analysis QMF bank and the output from the envelope adjuster. It first creates the matrix $\mathbf{X}$ from these outputs according to:

$$\mathbf{X}(k,l)=\begin{cases}\mathbf{X}_{Low}\left(k,l+t_{HFAdj}\right) & ,\ 0 \le k < k_x'+bsco',\ 0 \le l < l_{Temp} \\ \mathbf{Y}'\left(k,l+t_{HFAdj}+l_f\right) & ,\ k_x'+bsco' \le k < k_x'+M',\ 0 \le l < l_{Temp} \\ 0 & ,\ \max\left(k'_x+bsco',k'_x+M'\right) \le k < 64,\ 0 \le l < l_{Temp} \\ \mathbf{X}_{Low}\left(k,l+t_{HFAdj}\right) & ,\ 0 \le k < k_x+bsco,\ l_{Temp} \le l < l_f \\ \mathbf{Y}\left(k,l+t_{HFAdj}\right) & ,\ k_x+bsco \le k < k_x+M,\ l_{Temp} \le l < l_f \\ 0 & ,\ \max\left(k_x+bsco,k_x+M\right) \le k < 64,\ l_{Temp} \le l < l_f \end{cases}$$

where $l_f = numTimeSlots \cdot RATE$, and where $l_{Temp} = RATE \cdot \mathbf{t}_E'\left(L_E'\right) - numTimeSlots \cdot RATE$, and ‘ indicates the value of the previous SBR frame. At start-up l_{Temp} , k_x' and $bsco'$ are set to zero. Where $bsco = 0$ unless a scalable core coder is used, for which $bsco = \max\left(INT\left(maxAACLine \cdot 32 / frameLength\right) - k_x, 0\right)$, and where $bsco = \max\left(INT\left(maxAACLine \cdot 32 / frameLength\right) - lsb, 0\right)$, and where

$$maxAACLine = \begin{cases} swb\_offset\_long\_window\left[\max\left(max\_sfb-1,0\right)\right] & ,\ for\ long\ blocks \\ 8 \cdot swb\_offset\_short\_window\left[\max\left(max\_sfb-1,0\right)\right] & ,\ for\ short\ blocks \end{cases}$$

If the SBR tool is used for pure upsampling without SBR processing, the matrix $\mathbf{X}$ is created according to :

$$\mathbf{X}(k,l)=\begin{cases}\mathbf{X}_{Low}\left(k,l+t_{HFAdj}\right) & ,\ 0 \le k < 32,\ 0 \le l < numTimeSlots \cdot RATE \\ 0 & ,\ 32 \le k < 64,\ 0 \le l < numTimeSlots \cdot RATE \end{cases}$$

Subsequently, the subband sample matrix

$\mathbf{X}(k,\ l),\ 0 \le k < 64,\ 0 \le l < numTimeSlots \cdot RATE$,

is synthesized in the synthesis QMF bank in accordance to section 4.6.18.4.2. The resulting output samples are shown as the dashed block of Figure 4.45 (c) (Figure 4.45 (d) if downsampled SBR is used), where also the timing of the synthesis windows is shown.

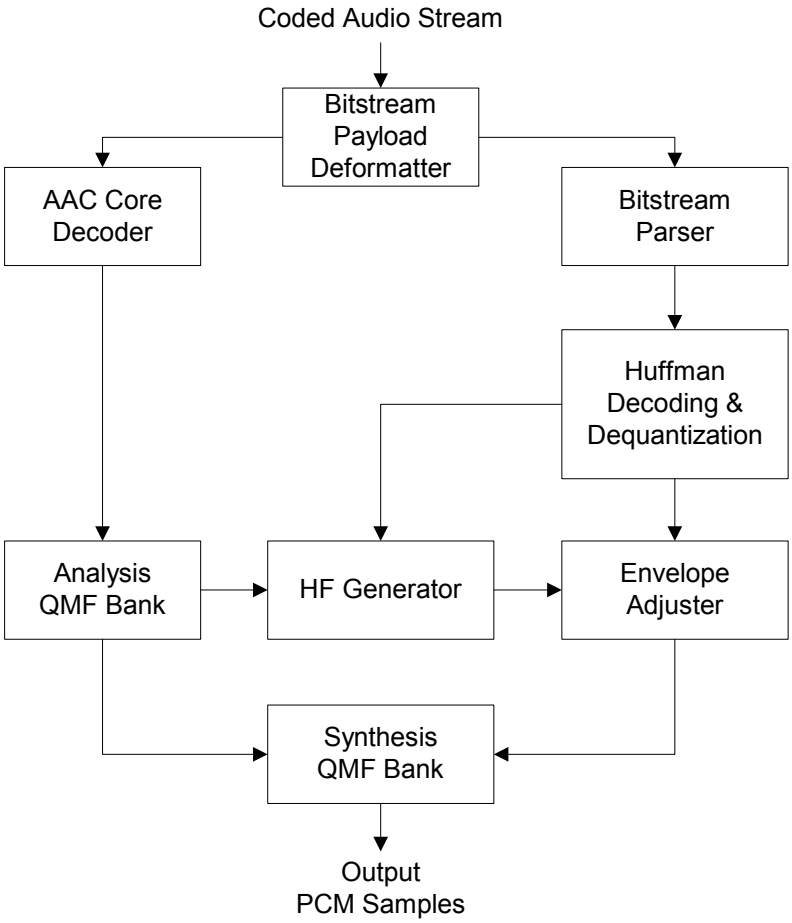
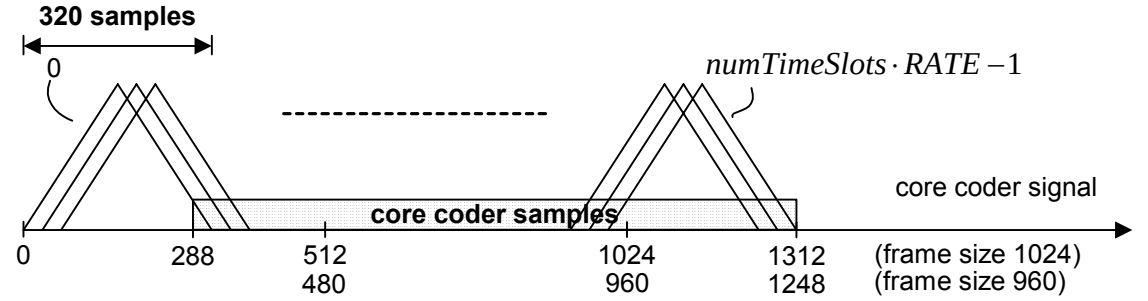
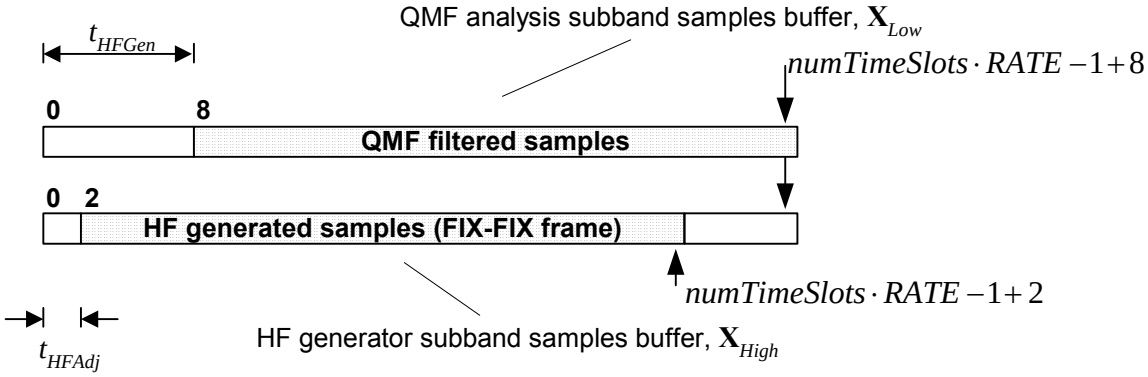


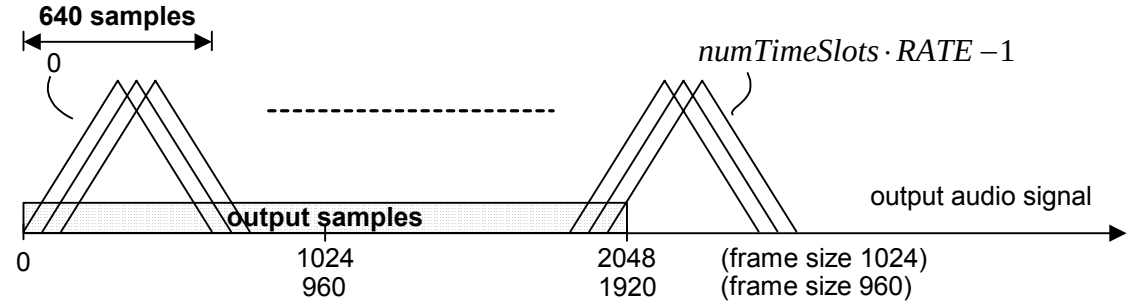
Figure 4.44 – Decoder block diagram



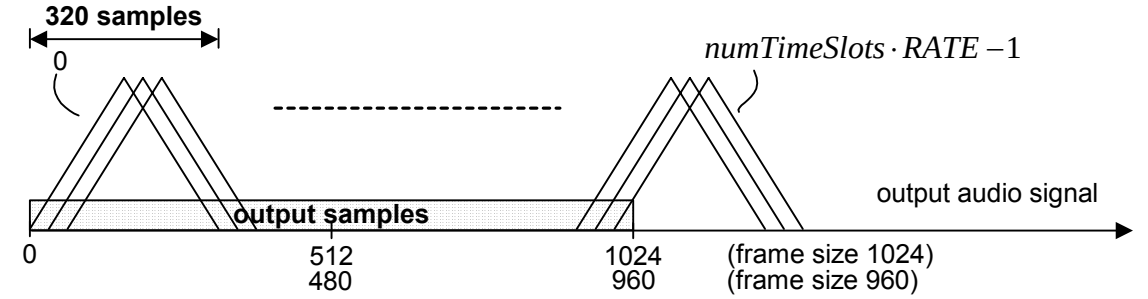
(a) core coder signal QMF analysis windowing



(b) subband samples buffers, $\mathbf{X}_{Low}$ and $\mathbf{X}_{High}$



(c) output signal QMF synthesis windowing



(d) output signal downsampled QMF synthesis windowing

Figure 4.45 – Synchronization and timing

4.6.18.6 HF generation

4.6.18.6.1 Introduction

The objective of the HF generator is to patch, or copy, a number of subband signals obtained from the analysis filterbank from consecutive subbands of matrix $\mathbf{X}_{Low}$ to consecutive subbands of matrix $\mathbf{X}_{High}$. The definition of the patching, i.e. the number of patches and the source ranges for the individual patches, is described by the vectors **patchNumSubbands** and **patchStartSubband**, and the variable *numPatches*. The subband signals $\mathbf{X}_{High}$ are inverse filtered according to the inverse filtering levels signaled from the encoder.

4.6.18.6.2 Inverse filtering

The inverse filtering is done in two steps. Linear prediction is first performed on the subband signals of $\mathbf{X}_{Low}$ Then the actual inverse filtering is done independently for each of the subband signals patched to $\mathbf{X}_{High}$ by the HF generator. The subband signals are complex valued, which results in complex filter coefficients for the linear prediction as well as for the inverse filtering. The prediction filter coefficients are obtained from the covariance method. The covariance matrix elements calculated are:

$$\phi_k(i,j) = \sum_{n=0}^{numTimeSlots \cdot RATE + 6 - 1} \mathbf{X}_{Low}\left(k, n-i+t_{HFAdj}\right) \cdot \mathbf{X}_{Low}{}^{*}\left(k, n-j+t_{HFAdj}\right) \quad , \begin{cases} 0 \le i < 3 \\ 1 \le j < 3 \\ 0 \le k < k_0 \end{cases}$$

The coefficients $\alpha_0(k)$ and $\alpha_1(k)$ used to filter the subband signal are calculated as:

$$\mathrm{d}(k) = \phi_k(2,2)\cdot\phi_k(1,1) - \frac{1}{1+\varepsilon_{Inv}}\left|\phi_k(1,2)\right|^2,$$

$$\alpha_1(k) = \begin{cases} \dfrac{\phi_k(0,1)\cdot\phi_k(1,2) - \phi_k(0,2)\cdot\phi_k(1,1)}{d(k)} & , d(k) \neq 0 \\ 0 & , d(k) = 0 \end{cases},$$

$$\alpha_0(k) = \begin{cases} -\dfrac{\phi_k(0,1) + \alpha_1(k)\cdot\phi_k^{*}(1,2)}{\phi_k(1,1)} & , \phi_k(1,1) \neq 0 \\ 0 & , \phi_k(1,1) = 0 \end{cases}.$$

In the first formula above ε_{Inv} is the relaxation parameter (ε_{Inv} = 1E-6). Moreover, if either of the magnitudes of $\alpha_0(k)$ and $\alpha_1(k)$ is greater than or equal to 4, both coefficients are set to zero.

The calculation of the chirp factors, **bwArray,** is shown below. Each chirp factor is used within a specific frequency range defined by the noise floor frequency band table, $\mathbf{f}_{TableNoise}$.

$$\mathbf{bwArray}(i) = \begin{cases} 0 & if\ \mathbf{tempBw}(i) < 0.015625 \\ \mathbf{tempBw}(i) & if\ \mathbf{tempBw}(i) \ge 0.015625 \end{cases}, \quad 0 \le i < N_Q$$

where $\mathbf{tempBw}(i)$ is calculated as

$$\mathbf{tempBw}(i) = \begin{cases} 0.75000\cdot newBw + 0.25000\cdot\mathbf{bwArray}'(i) & if\ newBw < \mathbf{bwArray}'(i) \\ 0.90625\cdot newBw + 0.09375\cdot\mathbf{bwArray}'(i) & if\ newBw \ge \mathbf{bwArray}'(i) \end{cases}, 0 \le i < N_Q$$

bwArray´ are the **bwArray** values calculated in the previous SBR frame, and are assumed to be zero for the first SBR frame. *newBw* is a function of **bs_invf_mode**(*i*) and **bs_invf_mode´**(*i*), given by Table 4.156, where **bs_invf_mode´** are the **bs_invf_mode** values from the previous SBR frame, and are assumed to be zero for the first frame.

Table 4.156 – *newBw* function

bs_invf_mode(*i*)´	**bs_invf_mode(*i*)**			
	Off	Low	Intermediate	Strong
Off	0.0	0.6	0.9	0.98
Low	0.6	0.75	0.9	0.98
Intermediate	0.0	0.75	0.9	0.98
Strong	0.0	0.75	0.9	0.98

4.6.18.6.3 HF generator

The patch is built in accordance to the flowchart of Figure 4.46, where the output variable *numPatches* is an integer value specifying the number of patches. **patchStartSubband** and **patchNumSubbands** are vectors holding the data output from the patch decision algorithm.

The HF generation is obtained according to:

$$\mathbf{X}_{High}\left(k, l + t_{HFAdj}\right) = \mathbf{X}_{Low}\left(p, l + t_{HFAdj}\right) + \mathbf{bwArray}\left(g\left(k\right)\right) \cdot \alpha_0\left(p\right) \cdot \mathbf{X}_{Low}\left(p, l - 1 + t_{HFAdj}\right) + \left[\mathbf{bwArray}\left(g\left(k\right)\right)\right]^2 \cdot \alpha_1\left(p\right) \cdot \mathbf{X}_{Low}\left(p, l - 2 + t_{HFAdj}\right),$$

where $g\left(k\right)$ is defined by $\mathbf{f}_{TableNoise}\left(g\left(k\right)\right) \le k < \mathbf{f}_{TableNoise}\left(g\left(k\right)+1\right)$ and

$$\begin{cases} k = k_x + x + \sum_{q=0}^{i-1} \mathbf{patchNumSubbands}\left(q\right) \\ p = \mathbf{patchStartSubband}\left(i\right) + x \end{cases}$$

for $0 \le x < \mathbf{patchNumSubbands}\left(i\right)$, $0 \le i < numPatches$, $RATE \cdot \mathbf{t}_E\left(0\right) \le l < RATE \cdot \mathbf{t}_E\left(L_E\right)$.

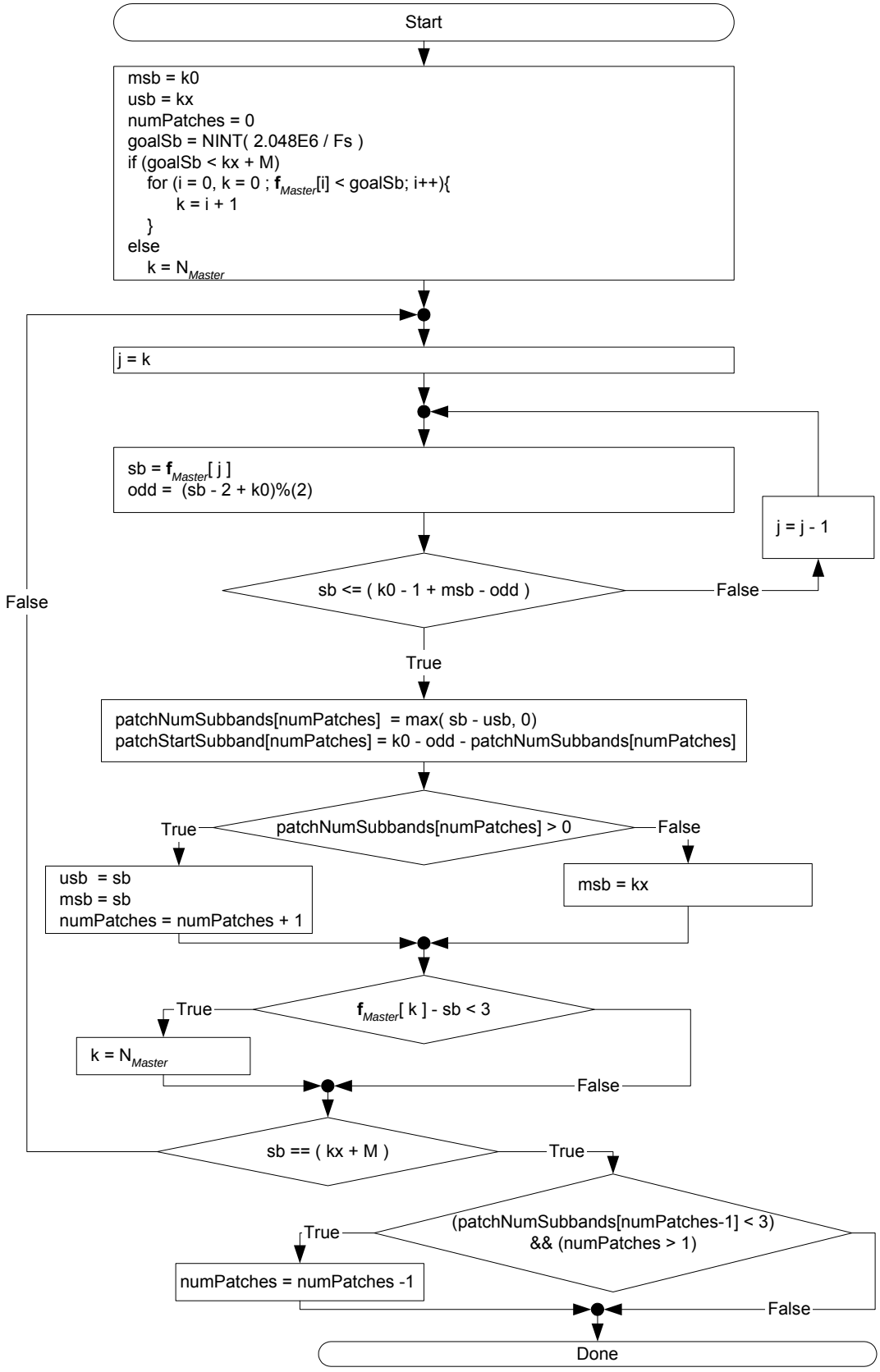


Figure 4.46 – Flowchart of patch construction

4.6.18.7 HF adjustment

4.6.18.7.1 Introduction

The envelope adjuster takes the input QMF-matrix $\mathbf{X}_{High}$ and produces the output QMF matrix $\mathbf{Y}$. The envelope adjustment is done upon the entire SBR range covering M QMF subbands, starting on subband k_x, for the time-frame spanned by the current SBR frame (indicated by the vector $\mathbf{t}_E$). Throughout the description below several temporary vectors and matrices are introduced in order to make the explanation stringent. All temporary matrices and vectors are indexed from zero, removing the k_x offset. The below description of the envelope adjustment is channel independent, and outlined for one channel only, and for one SBR frame only. Variables used below that originate from the processing of the previous SBR frame, are assumed to be zero for the first SBR frame.

4.6.18.7.2 Mapping

Some of the data extracted from the bitstream payload are vectors (or matrices) containing data elements representing a frequency range of several QMF subbands. In order to simplify the explanation below, and sometimes out of necessity, this grouped data is mapped to the highest available frequency resolution for the envelope adjustment, i.e. to the individual QMF subbands within the SBR range. This means that several adjacent subbands in the mapped vectors (or matrices) will have the same value.

The mapping of the envelope scalefactors and the noise floor scalefactors is outlined below. The SBR envelope is mapped to the resolution of the QMF bank, albeit with preserved time resolution. The noise floor scalefactors are also mapped to the frequency resolution of the filterbank, but with the time resolution of the envelope scalefactors.

$$\mathbf{E}_{OrigMapped}\left(m-k_x,l\right)=\mathbf{E}_{Orig}\left(i,l\right) \quad ,\mathbf{F}\left(i,\mathbf{r}\left(l\right)\right)\le m<\mathbf{F}\left(i+1,\mathbf{r}\left(l\right)\right),0\le i<\mathbf{n}\left(\mathbf{r}\left(l\right)\right),\ 0\le l<L_E$$

$$\mathbf{Q}_{Mapped}\left(m-k_x,l\right)=\mathbf{Q}_{Orig}\left(i,k\left(l\right)\right) \quad ,\mathbf{f}_{TableNoise}(i)\le m<\mathbf{f}_{TableNoise}(i+1),0\le i<N_Q,\ 0\le l<L_E$$

where $k\left(l\right)$ is defined by $RATE\cdot\mathbf{t}_E\left(l\right)\ge RATE\cdot\mathbf{t}_Q\left(k\left(l\right)\right), RATE\cdot\mathbf{t}_E\left(l+1\right)\le RATE\cdot\mathbf{t}_Q\left(k\left(l\right)+1\right)$, and $\mathbf{F}\left(i,\mathbf{r}\left(l\right)\right)$ is indexed as row, column i.e. $\mathbf{F}\left(i,\mathbf{r}\left(l\right)\right)$ gives $\mathbf{f}_{TableLow}\left(i\right)$ for $\mathbf{r}\left(l\right)=LO$ and $\mathbf{f}_{TableHigh}\left(i\right)$ for $\mathbf{r}\left(l\right)=HI$.

The mapping of the additional sinusoids is done below. In order to simplify two matrices are introduced, $\mathbf{S}_{IndexMapped}$ and $\mathbf{S}_{Mapped}$. The former is a binary matrix indicating in which QMF subbands sinusoids should be added, the latter is a matrix used to compensate the energy-values for the frequency bands where a sinusoid is added. If the bitstream payload indicates a sinusoid in a QMF subband where there was none present in the previous SBR frame, the generated sine should start at the position indicated by l_A (Table 4.157) in the present SBR frame. The generated sinusoid is placed in the middle of the high frequency resolution band, according to the below:

Let,

$$\mathbf{S}_{Index}\left(i\right)=\begin{cases}\mathbf{bs\_add\_harmonic}\left(i\right) & ,bs\_add\_harmonic\_flag=1\\ 0 & ,bs\_add\_harmonic\_flag=0\end{cases},0\le i<N_{High}$$

$$\mathbf{S}_{IndexMapped}\left(m-k_x,l\right)=\begin{cases}0 & if\ m\ne INT\left(\dfrac{\mathbf{f}_{TableHigh}\left(i+1\right)+\mathbf{f}_{TableHigh}\left(i\right)}{2}\right)\\ \mathbf{S}_{Index}\left(i\right)\cdot\delta_{Step}\left(m-k_x,l\right) & if\ m=INT\left(\dfrac{\mathbf{f}_{TableHigh}\left(i+1\right)+\mathbf{f}_{TableHigh}\left(i\right)}{2}\right)\end{cases}$$

for $\mathbf{f}_{TableHigh}\left(i\right)\le m<\mathbf{f}_{TableHigh}\left(i+1\right),\ 0\le i<N_{High}\ ,0\le l<L_E$

where

$$\delta_{Step}\left(m,l\right)=\begin{cases}1 & if\ \left(l\ge l_A\right)OR\left(\mathbf{S}'_{IndexMapped}\left(m,L'_E-1\right)=1\right)\\ 0 & otherwise\end{cases},$$

and where l_A is defined according to the table below,

Table 4.157 – Table for calculation of l_A

bs_pointer	*bs_frame_class*		
	FIXFIX	*FIXVAR,VARVAR*	*VARFIX*
= 0	-1	-1	-1
=1	-1	L_E+1-*bs_pointer*	-1
>1	-1	L_E+1-*bs_pointer*	*bs_pointer*-1

and $\mathbf{S}'_{IndexMapped}$ is $\mathbf{S}_{IndexMapped}$ of the previous SBR frame for the same frequency range. If the frequency range is larger for the current frame, the entries for the QMF subbands not covered by the previous $\mathbf{S}_{IndexMapped}$ are assumed to be zero.

The frequency resolution of the transmitted information on additional sinusoids is constant, therefore the varying frequency resolution of the envelope scalefactors needs to be considered. Since the frequency resolution of the envelope scalefactors is always coarser or as fine as that of the additional sinusoid data, the varying frequency resolution is handled according to the below:

$$\mathbf{S}_{Mapped}\left(m-k_x,l\right)=\delta_S\left(i,l\right),\ l_i\le m<u_i,\begin{cases}u_i=\mathbf{F}\left(i+1,\mathbf{r}\left(l\right)\right)\\ l_i=\mathbf{F}\left(i,\mathbf{r}\left(l\right)\right)\end{cases}$$

for $0\le i<\mathbf{n}\left(\mathbf{r}\left(l\right)\right),\ 0\le l<L_E$

where

$$\delta_S\left(i,l\right)=\begin{cases}1 & ,1\in\left\{\mathbf{S}_{IndexMapped}\left(j-k_x,l\right):\ \ \mathbf{F}\left(i,\mathbf{r}\left(l\right)\right)\le j<\mathbf{F}\left(i+1,\mathbf{r}\left(l\right)\right)\right\}\\ 0 & ,otherwise\end{cases}.$$

The $\delta_S\left(i,l\right)$ function returns one if any entry in the $\mathbf{S}_{IndexMapped}$ matrix is one within the given boundaries, i.e. if an additional sinusoid is present within the present frequency band. The $\mathbf{S}_{Mapped}$ matrix is hence one for all QMF subbands in the scalefactor bands where an additional sinusoid shall be added.

4.6.18.7.3 Estimation of current envelope

In order to envelope adjust the present SBR frame, the envelope of the current SBR signal needs to be estimated. This is done according to below, dependent on the data element *bs_interpol_freq*. The SBR envelope is estimated by averaging the squared complex subband samples over different time and frequency regions, given by the time/frequency grid represented by $\mathbf{t}_E$ and $\mathbf{r}$.

If interpolation (*bs_interpol_freq* = 1) is used:

$$\mathbf{E}_{Curr}\left(m,l\right)=\frac{\sum_{i=RATE\cdot\mathbf{t}_E\left(l\right)+t_{HFAdj}}^{RATE\cdot\mathbf{t}_E\left(l+1\right)-1+t_{HFAdj}}\left|\mathbf{X}_{High}\left(m+k_x,i\right)\right|^2}{\left(RATE\cdot\mathbf{t}_E\left(l+1\right)-RATE\cdot\mathbf{t}_E\left(l\right)\right)}\quad,\quad 0\le m<M,\ 0\le l<L_E$$

else, no interpolation (*bs_interpol_freq* = 0):

$$\mathbf{E}_{Curr}\left(k-k_x,l\right)=\frac{\sum_{i=RATE\cdot\mathbf{t}_E\left(l\right)+t_{HFAdj}}^{RATE\cdot\mathbf{t}_E\left(l+1\right)-1+t_{HFAdj}}\sum_{j=k_l}^{k_h}\left|\mathbf{X}_{High}\left(j,i\right)\right|^2}{\left(RATE\cdot\mathbf{t}_E\left(l+1\right)-RATE\cdot\mathbf{t}_E\left(l\right)\right)\cdot\left(k_h-k_l+1\right)},$$

$$k_l\le k\le k_h,\begin{cases}k_l=\mathbf{F}\left(p,\mathbf{r}\left(l\right)\right)\\ k_h=\mathbf{F}\left(p+1,\mathbf{r}\left(l\right)\right)-1\end{cases},0\le p<\mathbf{n}\left(\mathbf{r}\left(l\right)\right),0\le l<L_E$$

If interpolation is used, the energies are averaged over every QMF filterbank subband, else the energies are averaged over every frequency band. In either case, the energies are stored with the frequency resolution of the QMF filterbank. Hence the $\mathbf{E}_{Curr}$ matrix has L_E columns (one for every SBR envelope) and M rows (the number of QMF subbands covered by the SBR range).

4.6.18.7.4 Calculation of levels of additional HF signal components

The noise floor scalefactor is the ratio between the energy of the noise to be added to the envelope adjusted HF generated signal $\mathbf{X}_{High}$ and the energy of the same. Hence, in order to add the correct amount of noise, the noise floor scalefactor needs to be converted to a proper amplitude value, according to the following.

$$\mathbf{Q}_M\left(m,l\right)=\sqrt{\mathbf{E}_{OrigMapped}\left(m,l\right)\cdot\frac{\mathbf{Q}_{Mapped}\left(m,l\right)}{1+\mathbf{Q}_{Mapped}\left(m,l\right)}}\quad,\quad 0\le m<M,0\le l<L_E$$

The level of the sinusoids are derived from the SBR envelope scalefactors according to below.

$$\mathbf{S}_M\left(m,l\right)=\sqrt{\mathbf{E}_{OrigMapped}\left(m,l\right)\cdot\frac{\mathbf{S}_{IndexMapped}\left(m,l\right)}{1+\mathbf{Q}_{Mapped}\left(m,l\right)}}\quad,\quad 0\le m<M,0\le l<L_E$$

4.6.18.7.5 Calculation of gain

The gain to be applied for the subband samples in order to retain the correct envelope is calculated according to below. The level of additional sinusoids, as well as the level of the additional added noise, are taken into account.

$$\mathbf{G}\left(m,l\right)=\begin{cases}\sqrt{\frac{\mathbf{E}_{OrigMapped}\left(m,l\right)}{\left(\varepsilon+\mathbf{E}_{Curr}\left(m,l\right)\right)\cdot\left(1+\delta(l)\cdot\mathbf{Q}_{Mapped}(m,l)\right)}} & if\quad \mathbf{S}_{Mapped}\left(m,l\right)=0\\ \sqrt{\frac{\mathbf{E}_{OrigMapped}\left(m,l\right)}{\left(\varepsilon+\mathbf{E}_{Curr}\left(m,l\right)\right)}\cdot\frac{\mathbf{Q}_{Mapped}\left(m,l\right)}{\left(1+\mathbf{Q}_{Mapped}\left(m,l\right)\right)}} & if\quad \mathbf{S}_{Mapped}\left(m,l\right)\ne 0\end{cases},0\le m<M,0\le l<L_E$$

where

$$\delta\left(l\right)=\begin{cases}0 & if\ l=l_A\ OR\ l=l_{APrev}\\ 1 & otherwise\end{cases},$$

and where

$$l_{APrev}=\begin{cases}0 & if\ l'_A=L'_E\\ -1 & otherwise\end{cases}$$

is introduced, derived from l'_A and L'_E, which are the l_A and L_E values of the previous SBR frame.

In order to avoid unwanted noise substitution, the gain values are limited according to the following. Furthermore, the total level of a particular limiter band is adjusted in order to compensate for the energy-loss imposed by the limiter.

$$\mathbf{G}_{Max_{Temp}}(k,l) = \sqrt{\frac{\varepsilon_0 + \sum_{i=\mathbf{f}_{TableLim}(k)-k_x}^{\mathbf{f}_{TableLim}(k+1)-1-k_x} \mathbf{E}_{OrigMapped}(i,l)}{\varepsilon_0 + \sum_{i=\mathbf{f}_{TableLim}(k)-k_x}^{\mathbf{f}_{TableLim}(k+1)-1-k_x} \mathbf{E}_{Curr}(i,l)}} \cdot \mathbf{limGain}(bs\_limiter\_gains), \quad 0 \le k < N_L, 0 \le l < L_E$$

$$\mathbf{G}_{Max}(m,l) = \min\left(\mathbf{G}_{Max_{Temp}}(k(m),l), 10^5\right), \quad 0 \le m < M, 0 \le l < L_E$$

where $k(m)$ is defined by $\mathbf{f}_{TableLim}(k(m)) \le m + k_x < \mathbf{f}_{TableLim}(k(m)+1)$,

and where $\mathbf{limGain} = \left[0.70795, 1.0, 1.41254, 10^{10}\right]$, and where $\varepsilon_0 = 10^{-12}$.

The additional noise added to the HF generated signal is limited in proportion to the energy lost due to the limitation of the gain values, according to the following:

$$\mathbf{Q}_{M_{Lim}}(m,l) = \min\left(\mathbf{Q}_M(m,l), \mathbf{Q}_M(m,l) \cdot \frac{\mathbf{G}_{Max}(m,l)}{\mathbf{G}(m,l)}\right), \quad 0 \le m < M, 0 \le l < L_E$$

The gain values are limited according to the following:

$$\mathbf{G}_{Lim}(m,l) = \min\left(\mathbf{G}(m,l), \mathbf{G}_{Max}(m,l)\right), \quad 0 \le m < M, 0 \le l < L_E$$

As mentioned above, the limiter is compensated for by adjusting the total gain for a limiter band, in proportion to the lost energy due to limitation. This is calculated according to the following:

$$\mathbf{G}_{Boost_{Temp}}(k,l) = \sqrt{\frac{\varepsilon_0 + \sum_{i=\mathbf{f}_{TableLim}(k)-k_x}^{\mathbf{f}_{TableLim}(k+1)-1-k_x} \mathbf{E}_{OrigMapped}(i,l)}{\varepsilon_0 + \sum_{i=\mathbf{f}_{TableLim}(k)-k_x}^{\mathbf{f}_{TableLim}(k+1)-1-k_x} \left(\mathbf{E}_{Curr}(i,l) \cdot \mathbf{G}_{Lim}^2(i,l) + \mathbf{S}_M^2(i,l) + \delta(\mathbf{S}_M(i,l),l) \cdot \mathbf{Q}_{M_{Lim}}^2(i,l)\right)}}$$

for $0 \le k < N_L, 0 \le l < L_E$ where, $\delta(\mathbf{S}_M(i,l),l) = \begin{cases} 0 & , \mathbf{S}_M(i,l) \ne 0 \; OR \; l = l_A \; OR \; l = l_{APrev} \\ 1 & , otherwise \end{cases}$.

The compensation, or boost factor, is limited in order not to get too high energy values, according to:

$$\mathbf{G}_{Boost}(m,l) = \min\left(\mathbf{G}_{Boost_{Temp}}(k(m),l), 1.584893192\right), \quad 0 \le m < M, 0 \le l < L_E$$

where $k(m)$ is defined by $\mathbf{f}_{TableLim}(k(m)) \le m + k_x < \mathbf{f}_{TableLim}(k(m)+1)$, and where $\varepsilon_0 = 10^{-12}$.

This compensation is applied to the gain, the noise floor scalefactors and the sinusoid levels, according to below.

$$\mathbf{G}_{LimBoost}(m,l) = \mathbf{G}_{Lim}(m,l) \cdot \mathbf{G}_{Boost}(m,l), \quad 0 \le m < M, 0 \le l < L_E$$

$$\mathbf{Q}_{M_{Lim}Boost}(m,l) = \mathbf{Q}_{M_{Lim}}(m,l) \cdot \mathbf{G}_{Boost}(m,l), \quad 0 \le m < M, 0 \le l < L_E$$

$$\mathbf{S}_{MBoost}(m,l) = \mathbf{S}_M(m,l) \cdot \mathbf{G}_{Boost}(m,l), \quad 0 \le m < M, 0 \le l < L_E$$

4.6.18.7.6 Assembling HF signals

Analogous to the mapping of SBR envelope data and noise floor data to a higher time and frequency resolution, the gain values, representing a time-span of several QMF subsamples, are mapped to the highest time-resolution available for the envelope adjustment, i.e. to the individual QMF subsamples within the current SBR frame.

The gain values to be applied to the subband samples are smoothed using the filter $\mathbf{h}_{Smooth}$. The variable h_{SL} is used to control whether smoothing is applied or not, according to:

$h_{SL} = \begin{cases} 4 & , bs\_smoothing\_mode = 0 \\ 0 & , bs\_smoothing\_mode = 1 \end{cases}$ and the filter used is defined as following:

$$\mathbf{h}_{Smooth} = \begin{bmatrix} 0.33333333333333 \\ 0.30150283239582 \\ 0.21816949906249 \\ 0.11516383427084 \\ 0.03183050093751 \end{bmatrix}.$$

The smoothed gain values $\mathbf{G}_{Filt}$ are calculated according to the following equation:

$$\mathbf{G}_{Temp}\left(m, i + h_{SL}\right) = \mathbf{G}_{LimBoost}\left(m, l\right),\ RATE \cdot \mathbf{t}_E\left(l\right) \le i < RATE \cdot \mathbf{t}_E\left(l+1\right), 0 \le l < L_E, 0 \le m < M$$

$$\mathbf{G}_{Filt}\left(m,i\right) = \begin{cases} \sum_{j=0}^{h_{SL}} \mathbf{G}_{Temp}\left(m, i - j + h_{SL}\right) \cdot \mathbf{h}_{Smooth}\left(j\right) & if\ l \ne l_A\ AND\ h_{SL} \ne 0\ AND\ l \ne l_{APrev} \\ \mathbf{G}_{Temp}\left(m, i + h_{SL}\right) & otherwise \end{cases}$$

for $RATE \cdot \mathbf{t}_E\left(l\right) \le i < RATE \cdot \mathbf{t}_E\left(l+1\right), 0 \le m < M, 0 \le l < L_E$

.

The first h_{SL} columns of the $\mathbf{G}_{Temp}$ matrix are the last h_{SL} columns of the $\mathbf{G}_{Temp}$ matrix of the previous SBR frame, unless the reset-flag is set (*reset=1*) for which case the first h_{SL} columns of the $\mathbf{G}_{Temp}$ matrix are equal to $\mathbf{G}_{LimBoost}\left(m, 0\right)$ for all QMF subbands within the SBR range.

The smoothed gain values are applied to the input subband matrix $\mathbf{X}_{High}$, for all SBR envelopes of the current SBR frame, according to:

$$\mathbf{W}_1\left(m,i\right) = \mathbf{G}_{Filt}\left(m,i\right) \cdot \mathbf{X}_{High}\left(m + k_x, i + t_{HfAdj}\right),\quad RATE \cdot \mathbf{t}_E\left(0\right) \le i < RATE \cdot \mathbf{t}_E\left(L_E\right), 0 \le m < M$$

The level of the noise is smoothed similar to the smoothing of the gain values using the filter $\mathbf{h}_{Smooth}$ of length h_{SL}

$$\mathbf{Q}_{Temp}\left(m, i + h_{SL}\right) = \mathbf{Q}_{M_{Lim}Boost}\left(m, l\right),\ RATE \cdot \mathbf{t}_E\left(l\right) \le i < RATE \cdot \mathbf{t}_E\left(l+1\right), 0 \le l < L_E, 0 \le m < M$$

$$\mathbf{Q}_{Filt}\left(m,i\right) = \begin{cases} \mathbf{Q}_{Temp}\left(m,i\right) & if\ l \ne l_A\ AND\ \ l \ne l_{APrev}\ AND\ \mathbf{S}_{MBoost}\left(m,l\right) = 0\ AND\ h_{SL} = 0 \\ \sum_{j=0}^{h_{SL}} \mathbf{Q}_{Temp}\left(m, i - j + h_{SL}\right) \cdot \mathbf{h}_{Smooth}\left(j\right) & if\ l \ne l_A\ AND\ \ l \ne l_{APrev}\ AND\ \mathbf{S}_{MBoost}\left(m,l\right) = 0\ AND\ h_{SL} \ne 0 \\ 0 & otherwise \end{cases}$$

for $RATE \cdot \mathbf{t}_E\left(l\right) \le i < RATE \cdot \mathbf{t}_E\left(l+1\right), 0 \le m < M, 0 \le l < L_E$.

The first h_{SL} columns of the $\mathbf{Q}_{Temp}$ matrix are the last h_{SL} columns of the $\mathbf{Q}_{Temp}$ matrix of the previous SBR frame, unless the reset-flag is set (*reset=1*) for which case the first h_{SL} columns of the $\mathbf{Q}_{Temp}$ matrix are equal to $\mathbf{Q}_{M_{Lim}Boost}\left(m, 0\right)$ for all QMF subbands within the SBR range.

The noise, based on the noise table $\mathbf{V}$ (Table 4.A.88), is added to the output according to:

$$\begin{cases} Re\{\mathbf{W}_2(m,i)\} = Re\{\mathbf{W}_1(m,i)\} + \mathbf{Q}_{Filt}(m,i)\cdot\mathbf{V}(0, f_{IndexNoise}(i)) \\ Im\{\mathbf{W}_2(m,i)\} = Im\{\mathbf{W}_1(m,i)\} + \mathbf{Q}_{Filt}(m,i)\cdot\mathbf{V}(1, f_{IndexNoise}(i)) \end{cases}, \begin{cases} RATE\cdot\mathbf{t}_E(l) \le i < RATE\cdot\mathbf{t}_E(l+1), 0 \le l < L_E \\ 0 \le m < M \end{cases}$$

where

$$f_{IndexNoise}(i) = \left(index_{Noise} + \left(i - RATE\cdot\mathbf{t}_E(0)\right)\cdot M + m + 1\right) \bmod (512),$$

and $index_{Noise}$ is the last $f_{IndexNoise}$ from the previous SBR frame, unless the reset-flag is set (*reset=1*) for which case $index_{Noise} = 0$.

In the equation above, $\mathbf{V}\left(0, f_{IndexNoise(i)}\right) = \varphi_{\mathrm{Re},noise}\left(f_{IndexNoise}(i)\right)$ and $\mathbf{V}\left(1, f_{IndexNoise(i)}\right) = \varphi_{\mathrm{Im},noise}\left(f_{IndexNoise}(i)\right)$, where $\varphi_{\mathrm{Re},noise}(i)$ and $\varphi_{\mathrm{Im},noise}(i)$ are defined in Table 4.A.88.

The sinusoids are added at the level given by $\mathbf{S}_{MBoost}(m,l)$ for the QMF subbands indicated by $\mathbf{S}_{IndexMapped}(m,l)$. This gives the final output QMF matrix $\mathbf{Y}$, according to:

$$\begin{cases} Re\{\mathbf{Y}(m+k_x, i+t_{HFAdj})\} = Re\{\mathbf{W}_2(m,i)\} + \boldsymbol{\psi}_{Re}(m,l,i) \\ Im\{\mathbf{Y}(m+k_x, i+t_{HFAdj})\} = Im\{\mathbf{W}_2(m,i)\} + \boldsymbol{\psi}_{Im}(m,l,i) \end{cases}, \begin{cases} RATE\cdot\mathbf{t}_E(l) \le i < RATE\cdot\mathbf{t}_E(l+1), 0 \le l < L_E \\ 0 \le m < M \end{cases}$$

where

$$\boldsymbol{\psi}_{Re}(m,l,i) = \mathbf{S}_{MBoost}(m,l)\cdot\boldsymbol{\varphi}_{Re,sin}\left(f_{IndexSine}(i)\right)$$

$$\boldsymbol{\psi}_{Im}(m,l,i) = \mathbf{S}_{MBoost}(m,l)\cdot(-1)^{m+k_x}\cdot\boldsymbol{\varphi}_{Im,sin}\left(f_{IndexSine}(i)\right)$$

where $\boldsymbol{\varphi}$ and $f_{IndexSine}$ are defined below as:

$$\begin{cases} \boldsymbol{\varphi}_{Re,sin} = [1,0,-1,0] \\ \boldsymbol{\varphi}_{Im,sin} = [0,1,0,-1] \end{cases} \text{ and } f_{IndexSine}(i) = \left(index_{Sine} + i - RATE\cdot\mathbf{t}_E(0)\right) \bmod (4),$$

$index_{Sine} = (\text{the last } f_{IndexSine} \text{ from the previous SBR frame} + 1) \bmod (4)$, or $index_{Sine} = 0$ for the first frame.

4.6.18.8 Low power SBR tool

4.6.18.8.1 Introduction

This subclause outlines the differences for the implementation of the low power version of the SBR tool compared to the high quality version of the SBR tool outlined in subclauses 4.6.18.1 to 4.6.18.7. The low power SBR tool operates on real-valued signals, and hence a real-valued filterbank is used and all references to the imaginary part of variables in subclauses 4.6.18.1 to 4.6.18.7 should be ignored. Furthermore, the low power SBR tool incorporates additional modules in order to reduce aliasing introduced due to the real-valued processing.

In Figure 4.47 a block diagram of the low power SBR decoder displays how the additional modules are interconnected into the SBR decoder.

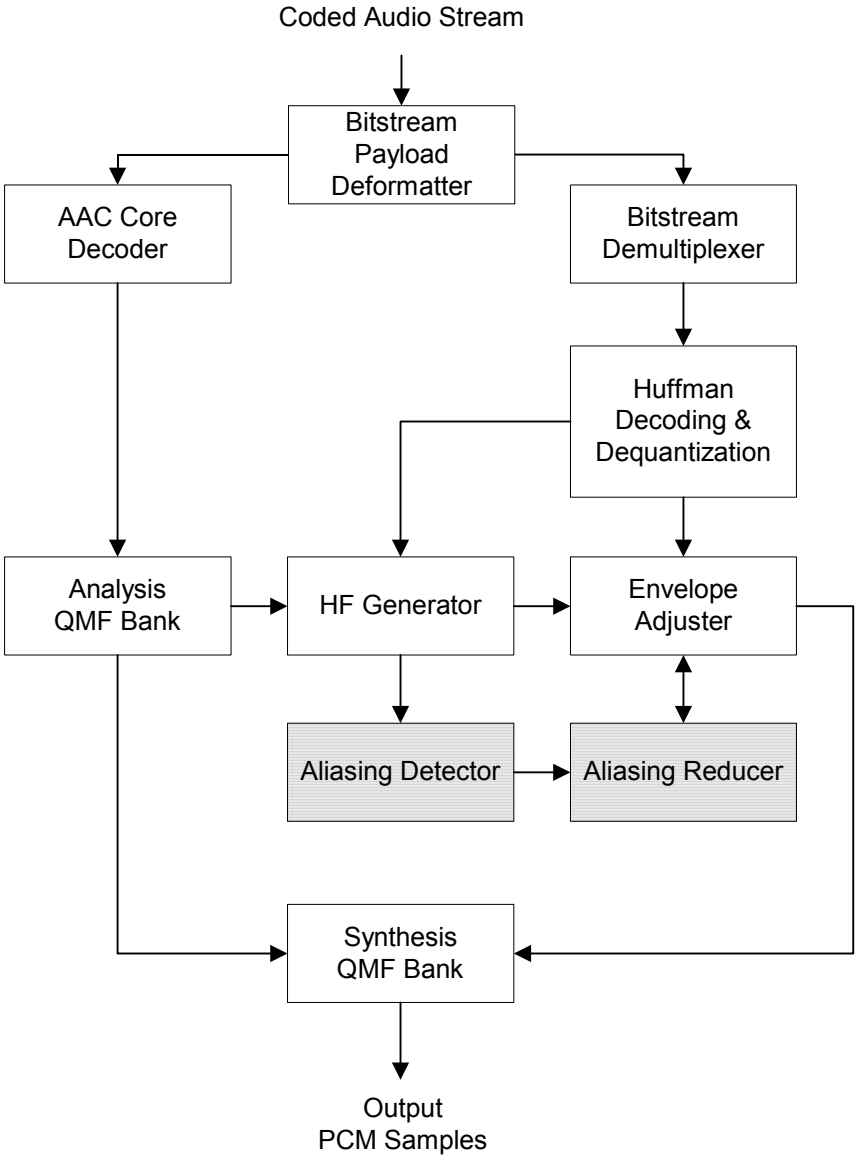


Figure 4.47 – Block diagram of the SBR low power decoder

4.6.18.8.2 Low power SBR tool filterbanks

4.6.18.8.2.1 Introduction

For the low power SBR tool, real-valued filterbanks are used. Hence, the filterbanks outlined in subclause 4.6.18.4 should be replaced by the following analysis and synthesis filterbanks.

4.6.18.8.2.2 Real-valued analysis filterbank

The real-valued QMF bank is used to split the time domain signal output from the core decoder into 32 subband signals. The output from the filterbank, i.e. the subband samples, are real-valued and critically sampled. The flowchart of the operation is given in Figure 4.48. The filtering involves the following steps, where an array $\mathbf{x}$ consisting of 320 time domain input samples is assumed. A higher index into the array corresponds to older samples.

- Shift the samples in the array $\mathbf{x}$ by 32 positions. The oldest 32 samples are discarded and 32 new samples are stored in positions 0 to 31.
- Multiply the samples of array $\mathbf{x}$ by every other coefficient of window $\mathbf{c}$. The window coefficients can be found in Table 4.A.87.
- Sum the samples according to the formula in the flowchart to create the 64-element array $\mathbf{u}$.
- Calculate new 32 subband samples by the matrix operation $\mathbf{M}_r\mathbf{u}$, where

$$\mathbf{M}_r(k,n) = 2 \cdot \cos\left(\frac{\pi \cdot (k+0.5) \cdot (2 \cdot n - 96)}{64}\right), \begin{cases} 0 \le k < 32 \\ 0 \le n < 64 \end{cases}$$

Every loop in the flowchart produces 32 subband samples, each representing the output from one filterbank subband. For every SBR frame the filterbank will produce $numTimeSlots \cdot RATE$ subband samples for every subband, corresponding to a time domain signal of length $numTimeSlots \cdot RATE \cdot 32$ samples. In the flowchart $\mathbf{W}$[k][l] corresponds to subband sample l of QMF subband k.

4.6.18.8.2.3 Real-valued synthesis filterbank

Synthesis filtering of the SBR-processed subband signals is achieved using a 64-subband QMF bank. The output from the filterbank are real-valued time domain samples. The process is given by the flowchart in Figure 4.49. The synthesis filtering comprises the following steps, where an array $\mathbf{v}$ consisting of 1280 samples is assumed:

- Shift the samples in the array $\mathbf{v}$ by 128 positions. The oldest 128 samples are discarded.
- The 64 new subband samples are multiplied by the matrix $\mathbf{N}_r$, where

$$\mathbf{N}_r(k,n) = \frac{1}{32} \cdot \cos\left(\frac{\pi \cdot (k+0.5) \cdot (2 \cdot n - 64)}{128}\right), \begin{cases} 0 \le k < 64 \\ 0 \le n < 128 \end{cases}$$

  The output from this operation is stored in the positions 0 to 127 of array $\mathbf{v}$.
- Extract samples from $\mathbf{v}$ according to the flowchart in Figure 4.49 to create the 640-element array $\mathbf{g}$.
- Multiply the samples of array $\mathbf{g}$ by window $\mathbf{c}$ to produce array $\mathbf{w}$. The window coefficients of $\mathbf{c}$ can be found in Table 4.A.87, and are the same as for the analysis filterbank.
- Calculate 64 new output samples by summation of samples from array $\mathbf{w}$ according to the formula in the flowchart of Figure 4.49.

Every SBR frame produces an output of $numTimeSlots \cdot RATE \cdot 64$ time domain samples. In the flowchart below $\mathbf{X}$[k][l] corresponds to subband sample l of QMF subband k, and every new loop produces 64 time domain samples as output.

4.6.18.8.2.4 Downsampled real-valued synthesis filterbank

Synthesis filtering of the SBR-processed subband signals is achieved using a 32-channel QMF bank. The output from the filterbank is real-valued time domain samples. The process is given by the flowchart in Figure 4.50. The synthesis filtering comprises the following steps, where an array **v** consisting of 640 samples is assumed:

- Shift the samples in the array **v** by 64 positions. The oldest 64 samples are discarded.
- The 32 new subband samples are multiplied by the matrix $\mathbf{N}_r$, where

$$\mathbf{N}_r(k,n) = \frac{1}{32} \cdot \cos\left(\frac{\pi \cdot (k + 0.5) \cdot (2 \cdot n - 32)}{64} \right) \quad , \begin{cases} 0 \le k < 32 \\ 0 \le n < 64 \end{cases}$$

  The output from this operation is stored in the positions 0 to 61 of array **v**.
- Extract samples from **v** according to the flowchart in Figure 4.50 to create the 320-element array **g**.
- Multiply the samples of array **g** by every other coefficient of window **c** to produce array **w**. The window coefficients of **c** can be found in Table 4.A.87, and are the same as for the analysis filterbank.
- Calculate 32 new output samples by summation of samples from array **w** according to the formula in the flowchart of Figure 4.50.

Every SBR frame produces an output of $numTimeSlots \cdot RATE \cdot 32$ time domain samples. In the flowchart of Figure 4.50 **X**[k][l] corresponds to subband sample l of QMF subband k, and every new loop produces 32 time domain samples as output.

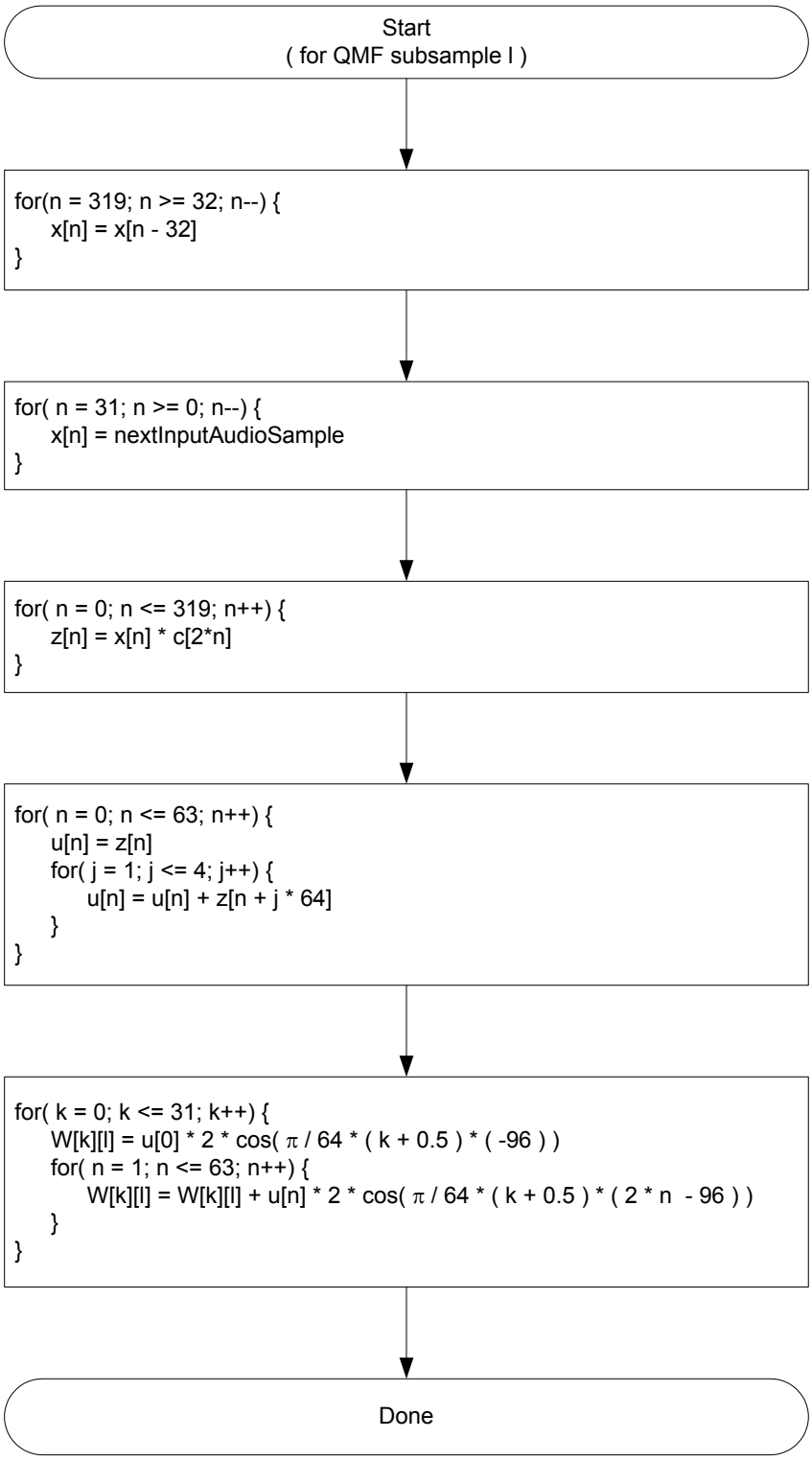


Figure 4.48 – Flowchart of decoder real-valued analysis QMF bank

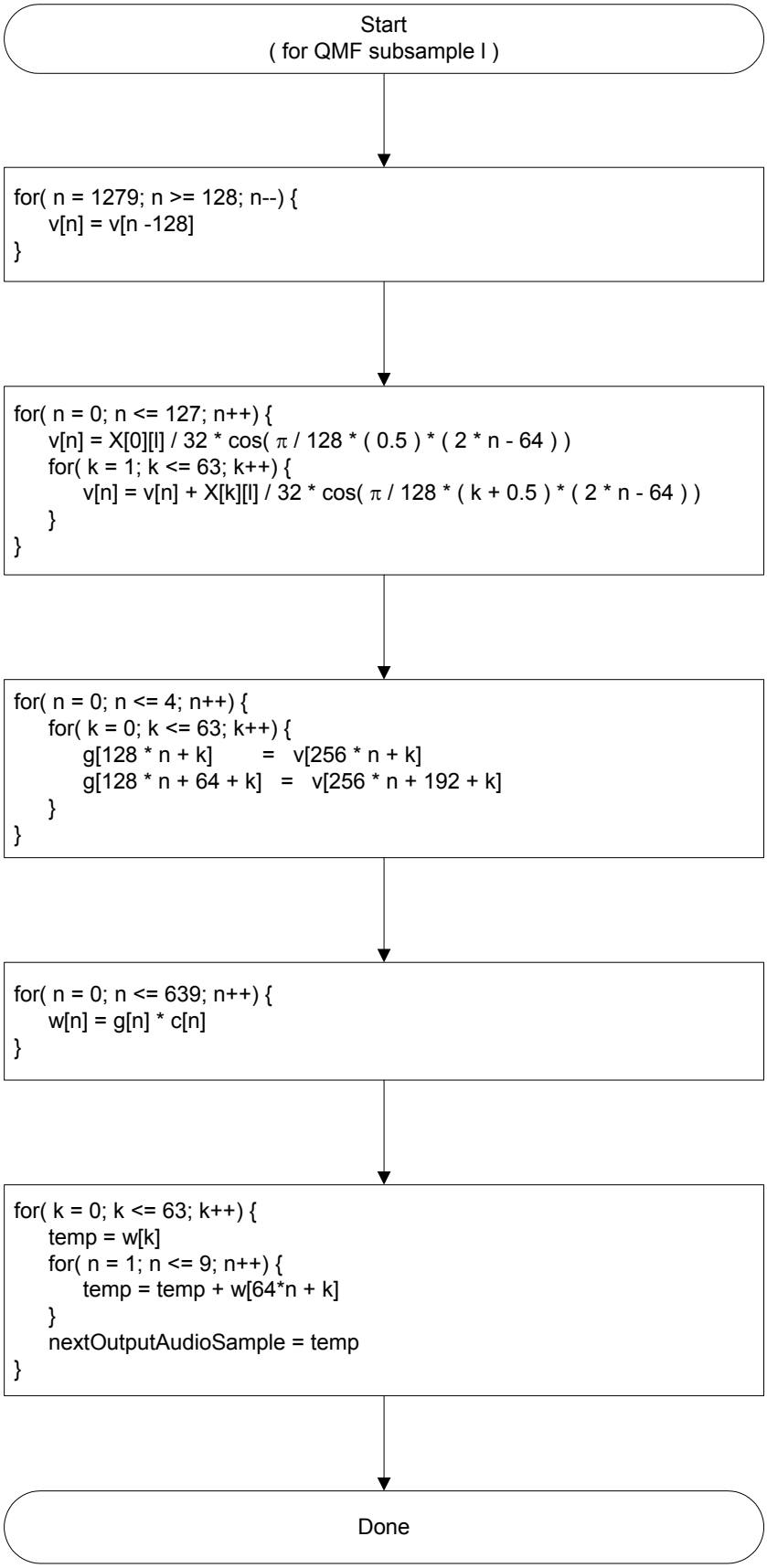


Figure 4.49 – Flowchart of decoder real-valued synthesis QMF bank

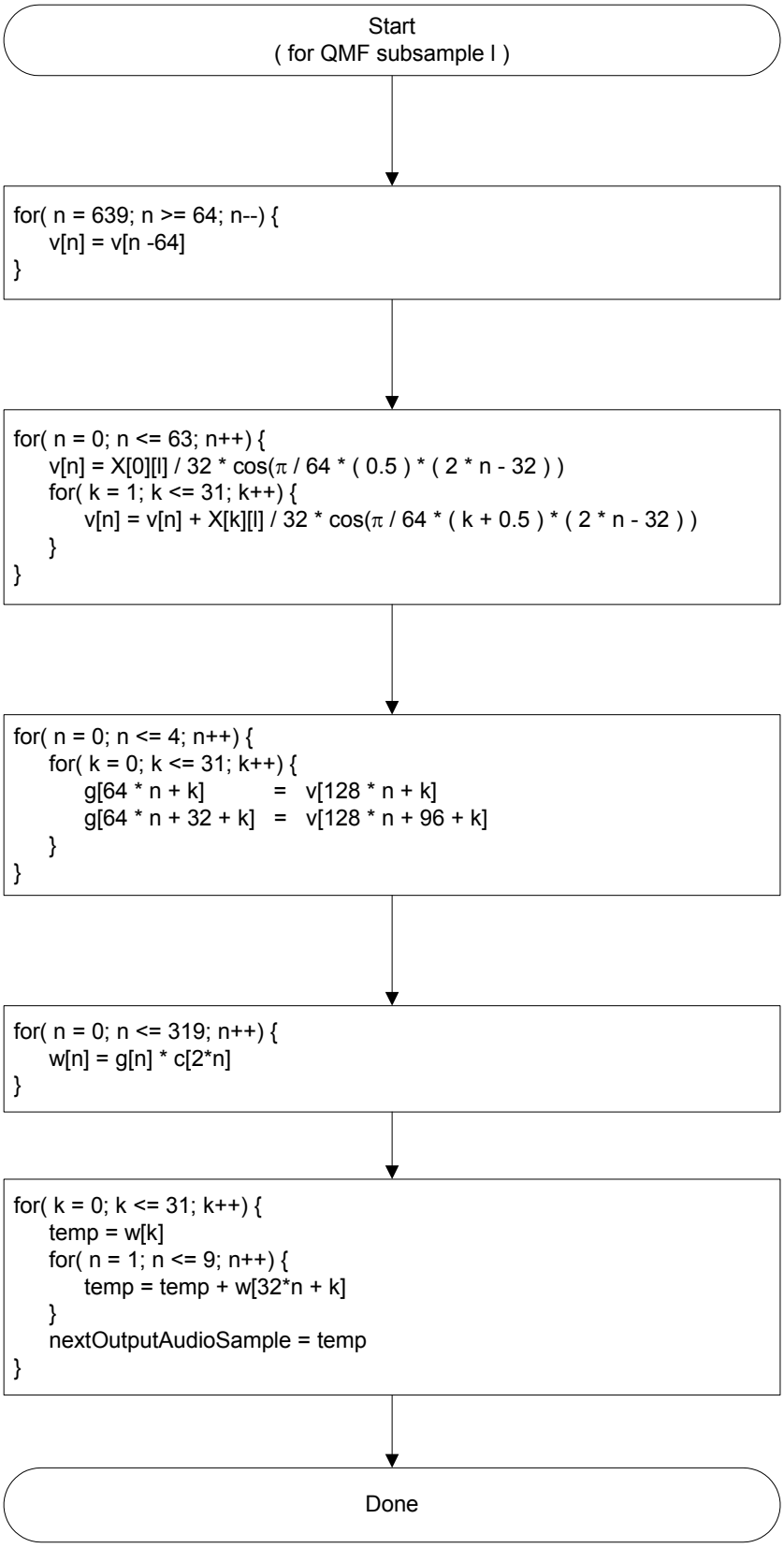


Figure 4.50 – Flowchart of decoder downsampled real-valued synthesis QMF bank

4.6.18.8.3 Aliasing detection

In order to minimize the introduction of aliasing by the envelope adjuster, the QMF subbands where strong aliasing will potentially be introduced are identified. The detection module uses data from the HF Generation module outlined in subclause 4.6.18.6, and from the HF Adjustment module outlined in subclause 4.6.18.7.

The aliasing detection algorithm calculates the reflection coefficient for every subband in the low-band.

$$\mathbf{ref}(k) = \begin{cases} \min\left(\max\left(-\dfrac{\phi_k(0,1)}{\phi_k(1,1)}, -1 \right), 1 \right) & \text{if } \phi_k(1,1) \neq 0 \\ 0 & \text{otherwise} \end{cases}, \quad 0 \leq k < k_0$$

Given the reflection coefficients **ref**, the degree of aliasing **deg** is calculated for the low-band according to the flowchart given in Figure 4.51.

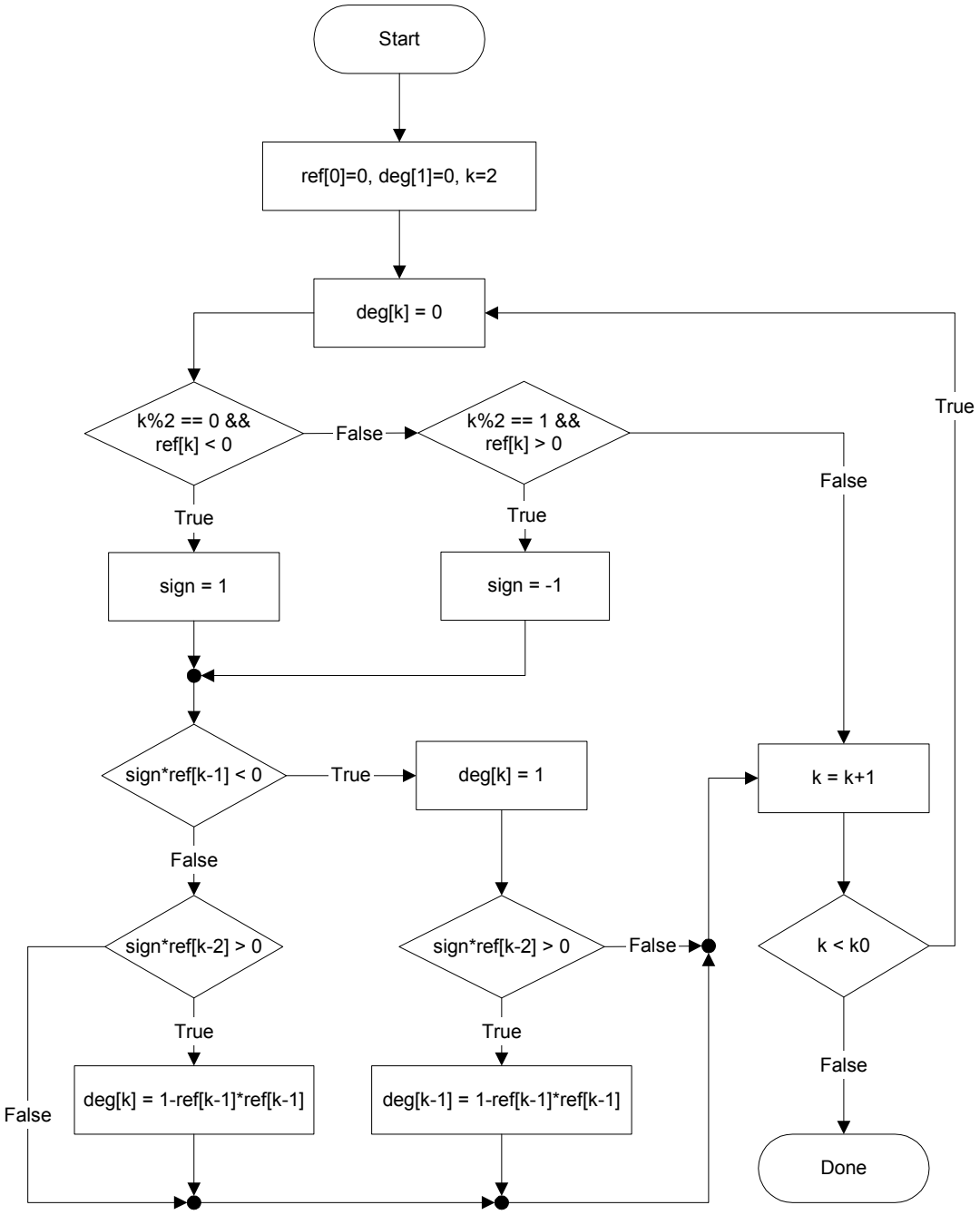


Figure 4.51 – Flowchart of the calculation of the aliasing degree

The degree of aliasing in the highband is obtained by using the patch information available in subclause 4.6.18.6, according to:

$$\mathbf{degPatched}(k) = \begin{cases} 0 & if\ x = 0 \\ \mathbf{deg}(p) & otherwise \end{cases}$$

where

$$\begin{cases} k = k_x + x + \sum_{q=0}^{i-1} \mathbf{patchNumSubbands}(q) \\ p = \mathbf{patchStartSubband}(i) + x \end{cases}, \quad 0 \le x < \mathbf{patchNumSubbands}(i), 0 \le i < numPatches \cdot$$

Since the patch information may not cover the whole SBR range, the degree of aliasing in the frequency region from where the patch ends to where the SBR range ends is defined by:

$$\mathbf{degPatched}(k) = 0, \quad k_x + \sum_{q=0}^{numPatches-1} \mathbf{patchNumSubbands}(q) \le k < k_x + M$$

Furthermore, the aliasing reduction algorithm needs a table to indicate the grouping of the gain-values. This table $\mathbf{F}_{Group}$ has L_E vectors of length $2 \cdot \mathbf{n}_G(l)$ representing the desired gain grouping for every SBR envelope of the SBR frame. It is calculated by the flowchart given in Figure 4.52 The table differs from previous tables in the text since it has individual start and stop indices for every group in frequency, whereas for the previous tables, the stop index of the previous group is taken as the start index of the current group. Hence, a vector representing N_G groups is $2N_G$ entries long, whereas a table of the style previously used would have been N_G +1 entries long. The stop index of a group is exclusive, i.e. the stop index subband is not included in the group.

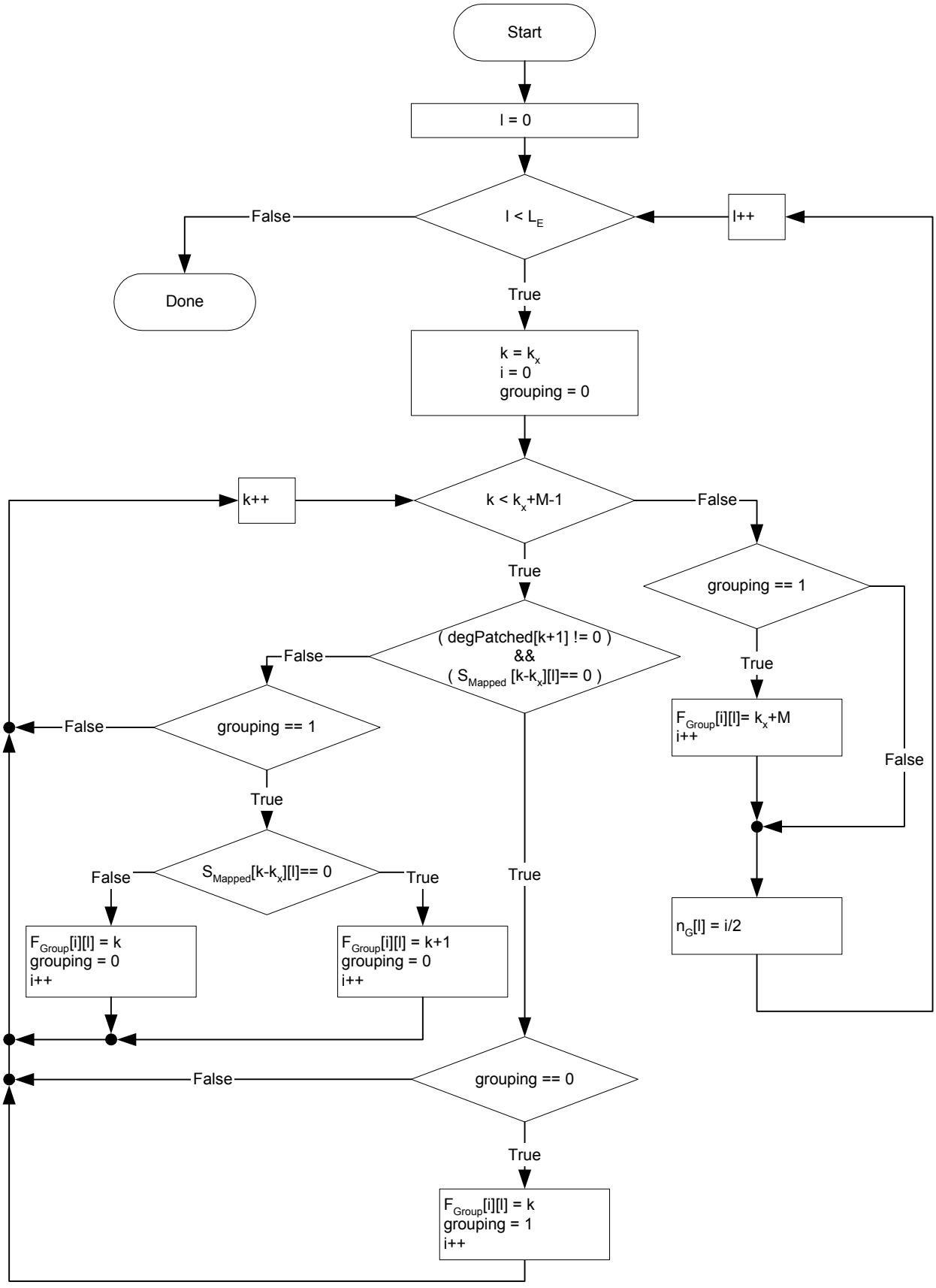


Figure 4.52 – Flowchart of the calculation of the gain groups

4.6.18.8.4 Modification of the energy calculation

Since the low power version of the SBR tool does not use complex-valued representation of signals, a modification of the energy calculation in subclause 4.6.18.7.3 is required. The equations given:

$$\mathbf{E}_{Curr}(m,l) = \frac{\sum_{i=RATE\cdot\mathbf{t}_E(l)+t_{HFAdj}}^{RATE\cdot\mathbf{t}_E(l+1)-1+t_{HFAdj}} \left|\mathbf{X}_{High}(m+k_x,i)\right|^2}{\left(RATE\cdot\mathbf{t}_E(l+1) - RATE\cdot\mathbf{t}_E(l)\right)} \quad , \quad 0 \le m < M, 0 \le l < L_E$$

and

$$\mathbf{E}_{Curr}(k-k_x,l) = \frac{\sum_{i=RATE\cdot\mathbf{t}_E(l)+t_{HFAdj}}^{RATE\cdot\mathbf{t}_E(l+1)-1+t_{HFAdj}} \sum_{j=k_l}^{k_h} \left|\mathbf{X}_{High}(j,i)\right|^2}{\left(RATE\cdot\mathbf{t}_E(l+1) - RATE\cdot\mathbf{t}_E(l)\right)\cdot(k_h - k_l + 1)}$$

is replaced by

$$\mathbf{E}_{Curr}(m,l) = \frac{2\cdot\sum_{i=RATE\cdot\mathbf{t}_E(l)+t_{HFAdj}}^{RATE\cdot\mathbf{t}_E(l+1)-1+t_{HFAdj}} \left|\mathbf{X}_{High}(m+k_x,i)\right|^2}{\left(RATE\cdot\mathbf{t}_E(l+1) - RATE\cdot\mathbf{t}_E(l)\right)} \quad , \quad 0 \le m < M, 0 \le l < L_E$$

and

$$\mathbf{E}_{Curr}(k-k_x,l) = \frac{2\cdot\sum_{i=RATE\cdot\mathbf{t}_E(l)+t_{HFAdj}}^{RATE\cdot\mathbf{t}_E(l+1)-1+t_{HFAdj}} \sum_{j=k_l}^{k_h} \left|\mathbf{X}_{High}(j,i)\right|^2}{\left(RATE\cdot\mathbf{t}_E(l+1) - RATE\cdot\mathbf{t}_E(l)\right)\cdot(k_h - k_l + 1)} .$$

4.6.18.8.5 Aliasing reduction

The aliasing reduction module re-calculates gain values calculated by the (real-valued) HF Adjustment module outlined in subclause 4.6.18.7. The variables available in the HF Adjustment subclause 4.6.18.7.5 are used by the aliasing reduction module outlined below. For the low power implementation, the output variable from the aliasing reduction module $\mathbf{G}_A$, as calculated below, should be used instead of $\mathbf{G}_{LimBoost}$ in the subsequent parts of the HF Adjustment module, i.e. subclause 4.6.18.7.6.

The energy of the subband signals in the affected subbands, if the calculated gain values $\mathbf{G}_{LimBoost}$ were used, would be:

$$\mathbf{E}_{Total}(k,l) = \sum_{i=\mathbf{F}_{Group}(2\cdot k,l)-k_x}^{\mathbf{F}_{Group}(2\cdot k+1,l)-1-k_x} \mathbf{G}^2_{LimBoost}(i,l)\cdot\mathbf{E}_{Curr}(i,l) \quad , \quad 0 \le k < \mathbf{n}_G(l), 0 \le l < L_E$$

Given this target energy $\mathbf{E}_{Total}$, a target gain value is calculated as follows:

$$\mathbf{G}^2_{Target}(k,l) = \frac{\mathbf{E}_{Total}(k,l)}{\varepsilon_0 + \sum_{i=\mathbf{F}_{Group}(2\cdot k,l)-k_x}^{\mathbf{F}_{Group}(2\cdot k+1,l)-1-k_x} \mathbf{E}_{Curr}(i,l)} \quad , \quad 0 \le k < \mathbf{n}_G(l), 0 \le l < L_E .$$

Given the above calculated target gain, a new gain value is calculated as a weighted sum of the original gain value and the newly calculated target gain:

$$\mathbf{G}^2_{ARtemp}(m-k_x,l) = \alpha(m)\cdot\mathbf{G}^2_{Target}(k,l) + \left(1-\alpha(m)\right)\cdot\mathbf{G}^2_{LimBoost}(m-k_x,l),$$
$$\mathbf{F}_{Group}(2\cdot k,l) \le m < \mathbf{F}_{Group}(2\cdot k+1,l)$$

$$0 \le k < \mathbf{n}_G(l), 0 \le l < L_E$$

where

$$\alpha(m) = \begin{cases} \max\left(\mathbf{degPatched}(m), \mathbf{degPatched}(m+1)\right) & ,if\ m < M + k_x - 1 \\ \mathbf{degPatched}(m) & ,if\ m = M + k_x - 1 \end{cases}$$

where $\mathbf{degPatched}(m)$, calculated in the aliasing detection part, is used as the degree of gain equalization between subband *m*-1 and subband *m*.

A new energy value is calculated based on the new gain values, according to:

$$\mathbf{E}_{TotalNew}(k,l) = \sum_{i=\mathbf{F}_{Group}(2\cdot k,l)-k_x}^{\mathbf{F}_{Group}(2\cdot k+1,l)-1-k_x} \mathbf{G}^2_{ARtemp}(i,l)\cdot \mathbf{E}_{Curr}(i,l) \quad , \quad 0 \le k < \mathbf{n}_G(l), 0 \le l < L_E$$

In order to retain the correct output energy, while limiting the gain-adjustment in order to avoid introduction of aliasing, the gain value $\mathbf{G}_A$ is calculated according to:

$$\mathbf{G}_A(m-k_x,l) = \begin{cases} \mathbf{G}_{ARtemp}(m-k_x,l)\cdot\sqrt{\dfrac{\mathbf{E}_{Total}(\kappa(m),l)}{\varepsilon_0 + \mathbf{E}_{TotalNew}(\kappa(m),l)}} & ,m \in A_G(l), \quad k_x \le m < k_x + M, 0 \le l < L_E \\ \mathbf{G}_{LimBoost}(m-k_x,l) & ,m \notin A_G(l) \end{cases}$$

where

$$A_G(l) = \bigcup_{k=0}^{\mathbf{n}_G(l)-1} \left\{ m : \mathbf{F}_{Group}(2\cdot k,l) \le m < \mathbf{F}_{Group}(2\cdot k+1,l) \right\},$$

and for $m \in A_G(l)$ define $\kappa(m)$ by $\mathbf{F}_{Group}(2\cdot\kappa(m),l) \le m < \mathbf{F}_{Group}(2\cdot\kappa(m)+1,l)$.

The $\mathbf{G}_A$ values are the new gain values that should be used instead of the $\mathbf{G}_{LimBoost}$ values in subclause 4.6.18.7.6.

For the low power version of the SBR tool, the gain smoothing process described in 4.6.18.7.6 is not applied regardless of the value of *bs_smoothing_mode*.

For the sinusoids added in subclause 4.6.18.7.6, modifications are required for the low-power version of the SBR tool. The following equations:

$$\begin{cases} Re\left\{\mathbf{Y}(m+k_x, i+t_{HFAdj})\right\} = Re\left\{\mathbf{W}_2(m,i)\right\} + \boldsymbol{\psi}_{Re}(m,l,i) \\ Im\left\{\mathbf{Y}(m+k_x, i+t_{HFAdj})\right\} = Im\left\{\mathbf{W}_2(m,i)\right\} + \boldsymbol{\psi}_{Im}(m,l,i) \end{cases}, \begin{cases} RATE\cdot\mathbf{t}_E(l) \le i < RATE\cdot\mathbf{t}_E(l+1), 0 \le l < L_E \\ 0 \le m < M \end{cases}$$

where

$$\boldsymbol{\psi}_{Re}(m,l,i) = \mathbf{S}_{MBoost}(m,l)\cdot\boldsymbol{\varphi}_{Re,sin}\left(f_{IndexSine}(i)\right)$$

$$\boldsymbol{\psi}_{Im}(m,l,i) = \mathbf{S}_{MBoost}(m,l)\cdot(-1)^{m+k_x}\cdot\boldsymbol{\varphi}_{Im,sin}\left(f_{IndexSine}(i)\right)$$

is replaced by:

$$Re\left\{\mathbf{Y}(m+k_x, i+t_{HFAdj})\right\} = \begin{cases} \boldsymbol{\psi}_m(m,l,i) & ,m = -1 \\ Re\left\{\mathbf{W}_2(m,i)\right\} + \boldsymbol{\psi}_m(m,l,i) & ,0 \le m < M \\ \boldsymbol{\psi}_m(m,l,i) & ,m = M\ AND\ m + k_x < 64 \end{cases}$$

where

$$-1 \le m \le M$$

$$RATE \cdot \mathbf{t}_E(l) \le i < RATE \cdot \mathbf{t}_E(l+1), 0 \le l < L_E$$

$$\boldsymbol{\psi}_m(m,l,i) = \boldsymbol{\psi}_{Re}(m,l,i) - 0.00815 \cdot (-1)^{m+k_x} \cdot \left(\boldsymbol{\psi}_{Re}(m-1,l,i-1) + \boldsymbol{\psi}_{Re}(m+1,l,i+1)\right)$$

$$\boldsymbol{\psi}_{Re}(m,l,i) = \mathbf{S}_{MBoost}(m,l) \cdot \boldsymbol{\varphi}_{Re,sin}\left(f_{IndexSine}(i)\right)$$

and where

$\mathbf{S}_{MBoost}(m,l) = 0$, for $m < 0$ or $m \ge M$.

The above modifications are only done for the first 16 (calculated in increasing frequency order) sinusoids, for every time segment.

Furthermore, since a signal, according to the above, may be added to $\mathbf{Y}(k_x - 1, i)$, i.e. the lowband, or $\mathbf{Y}(k_x + M, i)$, i.e. one QMF subband above the SBR range, the following equation in subclause 4.6.18.5 needs to be modified:

$$\mathbf{X}(k,l) = \begin{cases} \mathbf{X}_{Low}(k, l + t_{HFAdj}) & , 0 \le k < k_x' + bsco',\ 0 \le l < l_{Temp} \\ \mathbf{Y}'(k, l + t_{HFAdj} + l_f) & , k_x' + bsco' \le k < k_x' + M',\ 0 \le l < l_{Temp} \\ 0 & , \max(k'_x + bsco', k'_x + M') \le k < 64,\ 0 \le l < l_{Temp} \\ \mathbf{X}_{Low}(k, l + t_{HFAdj}) & , 0 \le k < k_x + bsco,\ l_{Temp} \le l < l_f \\ \mathbf{Y}(k, l + t_{HFAdj}) & , k_x + bsco \le k < k_x + M,\ l_{Temp} \le l < l_f \\ 0 & , \max(k_x + bsco, k_x + M) \le k < 64,\ l_{Temp} \le l < l_f \end{cases}$$

The above is replaced by

$$\mathbf{X}(k,l) = \begin{cases} \mathbf{X}_{Low}(k, l + t_{HFAdj}) & , 0 \le k < k_x' - 1 + bsco',\ 0 \le l < l_{Temp} \\ \mathbf{X}_{Low}(k, l + t_{HFAdj}) + \mathbf{Y}'(k, l + t_{HFAdj} + l_f) & , k = k_x' - 1 + bsco',\ 0 \le l < l_{Temp} \\ \mathbf{Y}'(k, l + t_{HFAdj} + l_f) & , k_x' + bsco' \le k \le \min(k_x' + M', 63),\ 0 \le l < l_{Temp} \\ 0 & , \begin{cases} \max(k_x' + bsco', k_x' + M') < k < 64 \\ 0 \le l < l_{Temp} \end{cases} \\ \mathbf{X}_{Low}(k, l + t_{HFAdj}) & , 0 \le k < k_x - 1 + bsco,\ l_{Temp} \le l < l_f \\ \mathbf{X}_{Low}(k, l + t_{HFAdj}) + \mathbf{Y}(k, l + t_{HFAdj}) & , k = k_x - 1 + bsco,\ l_{Temp} \le l < l_f \\ \mathbf{Y}(k, l + t_{HFAdj}) & , \begin{cases} k_x + bsco \le k \le \min(k_x + M, 63) \\ l_{Temp} \le l < l_f \end{cases} \\ 0 & , \begin{cases} \max(k_x + bsco, k_x + M) < k < 64 \\ l_{Temp} \le l < l_f \end{cases} \end{cases}$$

Annex 4.A
(normative)

Normative Tables

4.A.1 Huffman codebook tables for AAC-type noisless coding

Table 4.A.1 – Scalefactor Huffman Codebook

Index	length	codeword (hexadecimal)	Index	length	codeword (hexadecimal)
0	18	3ffe8	61	4	a
1	18	3ffe6	62	4	c
2	18	3ffe7	63	5	1b
3	18	3ffe5	64	6	39
4	19	7fff5	65	6	3b
5	19	7fff1	66	7	78
6	19	7ffed	67	7	7a
7	19	7fff6	68	8	f7
8	19	7ffee	69	8	f9
9	19	7ffef	70	9	1f6
10	19	7fff0	71	9	1f9
11	19	7fffc	72	10	3f4
12	19	7fffd	73	10	3f6
13	19	7ffff	74	10	3f8
14	19	7fffe	75	11	7f5
15	19	7fff7	76	11	7f4
16	19	7fff8	77	11	7f6
17	19	7fffb	78	11	7f7
18	19	7fff9	79	12	ff5
19	18	3ffe4	80	12	ff8
20	19	7fffa	81	13	1ff4
21	18	3ffe3	82	13	1ff6
22	17	1ffef	83	13	1ff8
23	17	1fff0	84	14	3ff8
24	16	fff5	85	14	3ff4
25	17	1ffee	86	16	fff0
26	16	fff2	87	15	7ff4
27	16	fff3	88	16	fff6
28	16	fff4	89	15	7ff5
29	16	fff1	90	18	3ffe2
30	15	7ff6	91	19	7ffd9
31	15	7ff7	92	19	7ffda
32	14	3ff9	93	19	7ffdb
33	14	3ff5	94	19	7ffdc
34	14	3ff7	95	19	7ffdd
35	14	3ff3	96	19	7ffde
36	14	3ff6	97	19	7ffd8
37	14	3ff2	98	19	7ffd2
38	13	1ff7	99	19	7ffd3
39	13	1ff5	100	19	7ffd4
40	12	ff9	101	19	7ffd5
41	12	ff7	102	19	7ffd6
42	12	ff6	103	19	7fff2

43	11	7f9	104	19	7ffdf
44	12	ff4	105	19	7ffe7
45	11	7f8	106	19	7ffe8
46	10	3f9	107	19	7ffe9
47	10	3f7	108	19	7ffea
48	10	3f5	109	19	7ffeb
49	9	1f8	110	19	7ffe6
50	9	1f7	111	19	7ffe0
51	8	fa	112	19	7ffe1
52	8	f8	113	19	7ffe2
53	8	f6	114	19	7ffe3
54	7	79	115	19	7ffe4
55	6	3a	116	19	7ffe5
56	6	38	117	19	7ffd7
57	5	1a	118	19	7ffec
58	4	b	119	19	7fff4
59	3	4	120	19	7fff3
60	1	0			

Table 4.A.2 – Spectrum Huffman Codebook 1

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	11	7f8	41	5	14
1	9	1f1	42	7	65
2	11	7fd	43	5	16
3	10	3f5	44	7	6d
4	7	68	45	9	1e9
5	10	3f0	46	7	63
6	11	7f7	47	9	1e4
7	9	1ec	48	7	6b
8	11	7f5	49	5	13
9	10	3f1	50	7	71
10	7	72	51	9	1e3
11	10	3f4	52	7	70
12	7	74	53	9	1f3
13	5	11	54	11	7fe
14	7	76	55	9	1e7
15	9	1eb	56	11	7f3
16	7	6c	57	9	1ef
17	10	3f6	58	7	60
18	11	7fc	59	9	1ee
19	9	1e1	60	11	7f0
20	11	7f1	61	9	1e2
21	9	1f0	62	11	7fa
22	7	61	63	10	3f3
23	9	1f6	64	7	6a
24	11	7f2	65	9	1e8
25	9	1ea	66	7	75
26	11	7fb	67	5	10
27	9	1f2	68	7	73
28	7	69	69	9	1f4
29	9	1ed	70	7	6e
30	7	77	71	10	3f7
31	5	17	72	11	7f6
32	7	6f	73	9	1e0

33	9	1e6	74	11	7f9
34	7	64	75	10	3f2
35	9	1e5	76	7	66
36	7	67	77	9	1f5
37	5	15	78	11	7ff
38	7	62	79	9	1f7
39	5	12	80	11	7f4
40	1	0			

Table 4.A.3 – Spectrum Huffman Codebook 2

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	9	1f3	41	5	7
1	7	6f	42	6	1d
2	9	1fd	43	5	b
3	8	eb	44	6	30
4	6	23	45	8	ef
5	8	ea	46	6	1c
6	9	1f7	47	7	64
7	8	e8	48	6	1e
8	9	1fa	49	5	c
9	8	f2	50	6	29
10	6	2d	51	8	f3
11	7	70	52	6	2f
12	6	20	53	8	f0
13	5	6	54	9	1fc
14	6	2b	55	7	71
15	7	6e	56	9	1f2
16	6	28	57	8	f4
17	8	e9	58	6	21
18	9	1f9	59	8	e6
19	7	66	60	8	f7
20	8	f8	61	7	68
21	8	e7	62	9	1f8
22	6	1b	63	8	ee
23	8	f1	64	6	22
24	9	1f4	65	7	65
25	7	6b	66	6	31
26	9	1f5	67	4	2
27	8	ec	68	6	26
28	6	2a	69	8	ed
29	7	6c	70	6	25
30	6	2c	71	7	6a
31	5	a	72	9	1fb
32	6	27	73	7	72
33	7	67	74	9	1fe
34	6	1a	75	7	69
35	8	f5	76	6	2e
36	6	24	77	8	f6
37	5	8	78	9	1ff
38	6	1f	79	7	6d
39	5	9	80	9	1f6
40	3	0			

Table 4.A.4 – Spectrum Huffman Codebook 3

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	1	0	41	10	3ef
1	4	9	42	9	1f3
2	8	ef	43	9	1f4
3	4	b	44	11	7f6
4	5	19	45	9	1e8
5	8	f0	46	10	3ea
6	9	1eb	47	13	1ffc
7	9	1e6	48	8	f2
8	10	3f2	49	9	1f1
9	4	a	50	12	ffb
10	6	35	51	10	3f5
11	9	1ef	52	11	7f3
12	6	34	53	12	ffc
13	6	37	54	8	ee
14	9	1e9	55	10	3f7
15	9	1ed	56	15	7ffe
16	9	1e7	57	9	1f0
17	10	3f3	58	11	7f5
18	9	1ee	59	15	7ffd
19	10	3ed	60	13	1ffb
20	13	1ffa	61	14	3ffa
21	9	1ec	62	16	ffff
22	9	1f2	63	8	f1
23	11	7f9	64	10	3f0
24	11	7f8	65	14	3ffc
25	10	3f8	66	9	1ea
26	12	ff8	67	10	3ee
27	4	8	68	14	3ffb
28	6	38	69	12	ff6
29	10	3f6	70	12	ffa
30	6	36	71	15	7ffc
31	7	75	72	11	7f2
32	10	3f1	73	12	ff5
33	10	3eb	74	16	fffe
34	10	3ec	75	10	3f4
35	12	ff4	76	11	7f7
36	5	18	77	15	7ffb
37	7	76	78	12	ff7
38	11	7f4	79	12	ff9
39	6	39	80	15	7ffa
40	7	74			

Table 4.A.5 – Spectrum Huffman Codebook 4

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	4	7	41	7	6b
1	5	16	42	8	e3
2	8	f6	43	7	69
3	5	18	44	9	1f3
4	4	8	45	8	eb
5	8	ef	46	8	e6

6	9	1ef	47	10	3f6
7	8	f3	48	7	6e
8	11	7f8	49	7	6a
9	5	19	50	9	1f4
10	5	17	51	10	3ec
11	8	ed	52	9	1f0
12	5	15	53	10	3f9
13	4	1	54	8	f5
14	8	e2	55	8	ec
15	8	f0	56	11	7fb
16	7	70	57	8	ea
17	10	3f0	58	7	6f
18	9	1ee	59	10	3f7
19	8	f1	60	11	7f9
20	11	7fa	61	10	3f3
21	8	ee	62	12	fff
22	8	e4	63	8	e9
23	10	3f2	64	7	6d
24	11	7f6	65	10	3f8
25	10	3ef	66	7	6c
26	11	7fd	67	7	68
27	4	5	68	9	1f5
28	5	14	69	10	3ee
29	8	f2	70	9	1f2
30	4	9	71	11	7f4
31	4	4	72	11	7f7
32	8	e5	73	10	3f1
33	8	f4	74	12	ffe
34	8	e8	75	10	3ed
35	10	3f4	76	9	1f1
36	4	6	77	11	7f5
37	4	2	78	11	7fe
38	8	e7	79	10	3f5
39	4	3	80	11	7fc
40	4	0			

Table 4.A.6 – Spectrum Huffman Codebook 5

index	**length**	**codeword** (hexadecimal)	**index**	**length**	**codeword** (hexadecimal)
0	13	1fff	41	4	a
1	12	ff7	42	7	71
2	11	7f4	43	8	f3
3	11	7e8	44	11	7e9
4	10	3f1	45	11	7ef
5	11	7ee	46	9	1ee
6	11	7f9	47	8	ef
7	12	ff8	48	5	18
8	13	1ffd	49	4	9
9	12	ffd	50	5	1b
10	11	7f1	51	8	eb
11	10	3e8	52	9	1e9
12	9	1e8	53	11	7ec
13	8	f0	54	11	7f6
14	9	1ec	55	10	3eb
15	10	3ee	56	9	1f3

16	11	7f2	57	8	ed
17	12	ffa	58	7	72
18	12	ff4	59	8	e9
19	10	3ef	60	9	1f1
20	9	1f2	61	10	3ed
21	8	e8	62	11	7f7
22	7	70	63	12	ff6
23	8	ec	64	11	7f0
24	9	1f0	65	10	3e9
25	10	3ea	66	9	1ed
26	11	7f3	67	8	f1
27	11	7eb	68	9	1ea
28	9	1eb	69	10	3ec
29	8	ea	70	11	7f8
30	5	1a	71	12	ff9
31	4	8	72	13	1ffc
32	5	19	73	12	ffc
33	8	ee	74	12	ff5
34	9	1ef	75	11	7ea
35	11	7ed	76	10	3f3
36	10	3f0	77	10	3f2
37	8	f2	78	11	7f5
38	7	73	79	12	ffb
39	4	b	80	13	1ffe
40	1	0			

Table 4.A.7 – Spectrum Huffman Codebook 6

index	Length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	11	7fe	41	4	3
1	10	3fd	42	6	2f
2	9	1f1	43	7	73
3	9	1eb	44	9	1fa
4	9	1f4	45	9	1e7
5	9	1ea	46	7	6e
6	9	1f0	47	6	2b
7	10	3fc	48	4	7
8	11	7fd	49	4	1
9	10	3f6	50	4	5
10	9	1e5	51	6	2c
11	8	ea	52	7	6d
12	7	6c	53	9	1ec
13	7	71	54	9	1f9
14	7	68	55	8	ee
15	8	f0	56	6	30
16	9	1e6	57	6	24
17	10	3f7	58	6	2a
18	9	1f3	59	6	25
19	8	ef	60	6	33
20	6	32	61	8	ec
21	6	27	62	9	1f2
22	6	28	63	10	3f8
23	6	26	64	9	1e4
24	6	31	65	8	ed
25	8	eb	66	7	6a

26	9	1f7	67	7	70
27	9	1e8	68	7	69
28	7	6f	69	7	74
29	6	2e	70	8	f1
30	4	8	71	10	3fa
31	4	4	72	11	7ff
32	4	6	73	10	3f9
33	6	29	74	9	1f6
34	7	6b	75	9	1ed
35	9	1ee	76	9	1f8
36	9	1ef	77	9	1e9
37	7	72	78	9	1f5
38	6	2d	79	10	3fb
39	4	2	80	11	7fc
40	4	0			

Table 4.A.8 – Spectrum Huffman Codebook 7

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	1	0	32	8	f3
1	3	5	33	8	ed
2	6	37	34	9	1e8
3	7	74	35	9	1ef
4	8	f2	36	10	3ef
5	9	1eb	37	10	3f1
6	10	3ed	38	10	3f9
7	11	7f7	39	11	7fb
8	3	4	40	9	1ed
9	4	c	41	8	ef
10	6	35	42	9	1ea
11	7	71	43	9	1f2
12	8	ec	44	10	3f3
13	8	ee	45	10	3f8
14	9	1ee	46	11	7f9
15	9	1f5	47	11	7fc
16	6	36	48	10	3ee
17	6	34	49	9	1ec
18	7	72	50	9	1f4
19	8	ea	51	10	3f4
20	8	f1	52	10	3f7
21	9	1e9	53	11	7f8
22	9	1f3	54	12	ffd
23	10	3f5	55	12	ffe
24	7	73	56	11	7f6
25	7	70	57	10	3f0
26	8	eb	58	10	3f2
27	8	f0	59	10	3f6
28	9	1f1	60	11	7fa
29	9	1f0	61	11	7fd
30	10	3ec	62	12	ffc
31	10	3fa	63	12	fff

Table 4.A.9 – Spectrum Huffman Codebook 8

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	5	e	32	7	71
1	4	5	33	6	2b
2	5	10	34	6	2d
3	6	30	35	6	31
4	7	6f	36	7	6d
5	8	f1	37	7	70
6	9	1fa	38	8	f2
7	10	3fe	39	9	1f9
8	4	3	40	8	ef
9	3	0	41	7	68
10	4	4	42	6	33
11	5	12	43	7	6b
12	6	2c	44	7	6e
13	7	6a	45	8	ee
14	7	75	46	8	f9
15	8	f8	47	10	3fc
16	5	f	48	9	1f8
17	4	2	49	7	74
18	4	6	50	7	73
19	5	14	51	8	ed
20	6	2e	52	8	f0
21	7	69	53	8	f6
22	7	72	54	9	1f6
23	8	f5	55	9	1fd
24	6	2f	56	10	3fd
25	5	11	57	8	f3
26	5	13	58	8	f4
27	6	2a	59	8	f7
28	6	32	60	9	1f7
29	7	6c	61	9	1fb
30	8	ec	62	9	1fc
31	8	fa	63	10	3ff

Table 4.A.10 – Spectrum Huffman Codebook 9

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	1	0	85	12	fda
1	3	5	86	12	fe3
2	6	37	87	12	fe9
3	8	e7	88	13	1fe6
4	9	1de	89	13	1ff3
5	10	3ce	90	13	1ff7
6	10	3d9	91	11	7d3
7	11	7c8	92	10	3d8
8	11	7cd	93	10	3e1
9	12	fc8	94	11	7d4
10	12	fdd	95	11	7d9
11	13	1fe4	96	12	fd3
12	13	1fec	97	12	fde
13	3	4	98	13	1fdd
14	4	c	99	13	1fd9
15	6	35	100	13	1fe2

16	7	72	101	13	1fea
17	8	ea	102	13	1ff1
18	8	ed	103	13	1ff6
19	9	1e2	104	11	7d2
20	10	3d1	105	10	3d4
21	10	3d3	106	10	3da
22	10	3e0	107	11	7c7
23	11	7d8	108	11	7d7
24	12	fcf	109	11	7e2
25	12	fd5	110	12	fce
26	6	36	111	12	fdb
27	6	34	112	13	1fd8
28	7	71	113	13	1fee
29	8	e8	114	14	3ff0
30	8	ec	115	13	1ff4
31	9	1e1	116	14	3ff2
32	10	3cf	117	11	7e1
33	10	3dd	118	10	3df
34	10	3db	119	11	7c9
35	11	7d0	120	11	7d6
36	12	fc7	121	12	fca
37	12	fd4	122	12	fd0
38	12	fe4	123	12	fe5
39	8	e6	124	12	fe6
40	7	70	125	13	1feb
41	8	e9	126	13	1fef
42	9	1dd	127	14	3ff3
43	9	1e3	128	14	3ff4
44	10	3d2	129	14	3ff5
45	10	3dc	130	12	fe0
46	11	7cc	131	11	7ce
47	11	7ca	132	11	7d5
48	11	7de	133	12	fc6
49	12	fd8	134	12	fd1
50	12	fea	135	12	fe1
51	13	1fdb	136	13	1fe0
52	9	1df	137	13	1fe8
53	8	eb	138	13	1ff0
54	9	1dc	139	14	3ff1
55	9	1e6	140	14	3ff8
56	10	3d5	141	14	3ff6
57	10	3de	142	15	7ffc
58	11	7cb	143	12	fe8
59	11	7dd	144	11	7df
60	11	7dc	145	12	fc9
61	12	fcd	146	12	fd7
62	12	fe2	147	12	fdc
63	12	fe7	148	13	1fdc
64	13	1fe1	149	13	1fdf
65	10	3d0	150	13	1fed
66	9	1e0	151	13	1ff5
67	9	1e4	152	14	3ff9
68	10	3d6	153	14	3ffb
69	11	7c5	154	15	7ffd
70	11	7d1	155	15	7ffe
71	11	7db	156	13	1fe7

72	12	fd2	157	12	fcc
73	11	7e0	158	12	fd6
74	12	fd9	159	12	fdf
75	12	feb	160	13	1fde
76	13	1fe3	161	13	1fda
77	13	1fe9	162	13	1fe5
78	11	7c4	163	13	1ff2
79	9	1e5	164	14	3ffa
80	10	3d7	165	14	3ff7
81	11	7c6	166	14	3ffc
82	11	7cf	167	14	3ffd
83	11	7da	168	15	7fff
84	12	fcb			

Table 4.A.11 – Spectrum Huffman Codebook 10

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	6	22	85	9	1c7
1	5	8	86	9	1ca
2	6	1d	87	9	1e0
3	6	26	88	10	3db
4	7	5f	89	10	3e8
5	8	d3	90	11	7ec
6	9	1cf	91	9	1e3
7	10	3d0	92	8	d2
8	10	3d7	93	8	cb
9	10	3ed	94	8	d0
10	11	7f0	95	8	d7
11	11	7f6	96	8	db
12	12	ffd	97	9	1c6
13	5	7	98	9	1d5
14	4	0	99	9	1d8
15	4	1	100	10	3ca
16	5	9	101	10	3da
17	6	20	102	11	7ea
18	7	54	103	11	7f1
19	7	60	104	9	1e1
20	8	d5	105	8	d4
21	8	dc	106	8	cf
22	9	1d4	107	8	d6
23	10	3cd	108	8	de
24	10	3de	109	8	e1
25	11	7e7	110	9	1d0
26	6	1c	111	9	1d6
27	4	2	112	10	3d1
28	5	6	113	10	3d5
29	5	c	114	10	3f2
30	6	1e	115	11	7ee
31	6	28	116	11	7fb
32	7	5b	117	10	3e9
33	8	cd	118	9	1cd
34	8	d9	119	9	1c8
35	9	1ce	120	9	1cb
36	9	1dc	121	9	1d1
37	10	3d9	122	9	1d7

38	10	3f1	123	9	1df
39	6	25	124	10	3cf
40	5	b	125	10	3e0
41	5	a	126	10	3ef
42	5	d	127	11	7e6
43	6	24	128	11	7f8
44	7	57	129	12	ffa
45	7	61	130	10	3eb
46	8	cc	131	9	1dd
47	8	dd	132	9	1d3
48	9	1cc	133	9	1d9
49	9	1de	134	9	1db
50	10	3d3	135	10	3d2
51	10	3e7	136	10	3cc
52	7	5d	137	10	3dc
53	6	21	138	10	3ea
54	6	1f	139	11	7ed
55	6	23	140	11	7f3
56	6	27	141	11	7f9
57	7	59	142	12	ff9
58	7	64	143	11	7f2
59	8	d8	144	10	3ce
60	8	df	145	9	1e4
61	9	1d2	146	10	3cb
62	9	1e2	147	10	3d8
63	10	3dd	148	10	3d6
64	10	3ee	149	10	3e2
65	8	d1	150	10	3e5
66	7	55	151	11	7e8
67	6	29	152	11	7f4
68	7	56	153	11	7f5
69	7	58	154	11	7f7
70	7	62	155	12	ffb
71	8	ce	156	11	7fa
72	8	e0	157	10	3ec
73	8	e2	158	10	3df
74	9	1da	159	10	3e1
75	10	3d4	160	10	3e4
76	10	3e3	161	10	3e6
77	11	7eb	162	10	3f0
78	9	1c9	163	11	7e9
79	7	5e	164	11	7ef
80	7	5a	165	12	ff8
81	7	5c	166	12	ffe
82	7	63	167	12	ffc
83	8	ca	168	12	fff
84	8	da			

Table 4.A.12 – Spectrum Huffman Codebook 11

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	4	0	145	10	38d
1	5	6	146	10	398
2	6	19	147	10	3b7
3	7	3d	148	10	3d3

4	8	9c	149	10	3d1
5	8	c6	150	10	3db
6	9	1a7	151	11	7dd
7	10	390	152	8	b4
8	10	3c2	153	10	3de
9	10	3df	154	9	1a9
10	11	7e6	155	9	19b
11	11	7f3	156	9	19c
12	12	ffb	157	9	1a1
13	11	7ec	158	9	1aa
14	12	ffa	159	9	1ad
15	12	ffe	160	9	1b3
16	10	38e	161	10	38b
17	5	5	162	10	3b2
18	4	1	163	10	3b8
19	5	8	164	10	3ce
20	6	14	165	10	3e1
21	7	37	166	10	3e0
22	7	42	167	11	7d2
23	8	92	168	11	7e5
24	8	af	169	8	b7
25	9	191	170	11	7e3
26	9	1a5	171	9	1bb
27	9	1b5	172	9	1a8
28	10	39e	173	9	1a6
29	10	3c0	174	9	1b0
30	10	3a2	175	9	1b2
31	10	3cd	176	9	1b7
32	11	7d6	177	10	39b
33	8	ae	178	10	39a
34	6	17	179	10	3ba
35	5	7	180	10	3b5
36	5	9	181	10	3d6
37	6	18	182	11	7d7
38	7	39	183	10	3e4
39	7	40	184	11	7d8
40	8	8e	185	11	7ea
41	8	a3	186	8	ba
42	8	b8	187	11	7e8
43	9	199	188	10	3a0
44	9	1ac	189	9	1bd
45	9	1c1	190	9	1b4
46	10	3b1	191	10	38a
47	10	396	192	9	1c4
48	10	3be	193	10	392
49	10	3ca	194	10	3aa
50	8	9d	195	10	3b0
51	7	3c	196	10	3bc
52	6	15	197	10	3d7
53	6	16	198	11	7d4
54	6	1a	199	11	7dc
55	7	3b	200	11	7db
56	7	44	201	11	7d5
57	8	91	202	11	7f0
58	8	a5	203	8	c1
59	8	be	204	11	7fb

60	9	196	205	10	3c8
61	9	1ae	206	10	3a3
62	9	1b9	207	10	395
63	10	3a1	208	10	39d
64	10	391	209	10	3ac
65	10	3a5	210	10	3ae
66	10	3d5	211	10	3c5
67	8	94	212	10	3d8
68	8	9a	213	10	3e2
69	7	36	214	10	3e6
70	7	38	215	11	7e4
71	7	3a	216	11	7e7
72	7	41	217	11	7e0
73	8	8c	218	11	7e9
74	8	9b	219	11	7f7
75	8	b0	220	9	190
76	8	c3	221	11	7f2
77	9	19e	222	10	393
78	9	1ab	223	9	1be
79	9	1bc	224	9	1c0
80	10	39f	225	10	394
81	10	38f	226	10	397
82	10	3a9	227	10	3ad
83	10	3cf	228	10	3c3
84	8	93	229	10	3c1
85	8	bf	230	10	3d2
86	7	3e	231	11	7da
87	7	3f	232	11	7d9
88	7	43	233	11	7df
89	7	45	234	11	7eb
90	8	9e	235	11	7f4
91	8	a7	236	11	7fa
92	8	b9	237	9	195
93	9	194	238	11	7f8
94	9	1a2	239	10	3bd
95	9	1ba	240	10	39c
96	9	1c3	241	10	3ab
97	10	3a6	242	10	3a8
98	10	3a7	243	10	3b3
99	10	3bb	244	10	3b9
100	10	3d4	245	10	3d0
101	8	9f	246	10	3e3
102	9	1a0	247	10	3e5
103	8	8f	248	11	7e2
104	8	8d	249	11	7de
105	8	90	250	11	7ed
106	8	98	251	11	7f1
107	8	a6	252	11	7f9
108	8	b6	253	11	7fc
109	8	c4	254	9	193
110	9	19f	255	12	ffd
111	9	1af	256	10	3dc
112	9	1bf	257	10	3b6
113	10	399	258	10	3c7
114	10	3bf	259	10	3cc
115	10	3b4	260	10	3cb

116	10	3c9	261	10	3d9
117	10	3e7	262	10	3da
118	8	a8	263	11	7d3
119	9	1b6	264	11	7e1
120	8	ab	265	11	7ee
121	8	a4	266	11	7ef
122	8	aa	267	11	7f5
123	8	b2	268	11	7f6
124	8	c2	269	12	ffc
125	8	c5	270	12	fff
126	9	198	271	9	19d
127	9	1a4	272	9	1c2
128	9	1b8	273	8	b5
129	10	38c	274	8	a1
130	10	3a4	275	8	96
131	10	3c4	276	8	97
132	10	3c6	277	8	95
133	10	3dd	278	8	99
134	10	3e8	279	8	a0
135	8	ad	280	8	a2
136	10	3af	281	8	ac
137	9	192	282	8	a9
138	8	bd	283	8	b1
139	8	bc	284	8	b3
140	9	18e	285	8	bb
141	9	197	286	8	c0
142	9	19a	287	9	18f
143	9	1a3	288	5	4
144	9	1b1			

4.A.2 Window tables

Table 4.A.13 – Kaiser-Bessel window for AAC SSR object type EIGHT_SHORT_SEQUENCE

i	*w(i)*	*i*	*w(I)*
0	0.0000875914060105	16	0.7446454751465113
1	0.0009321760265333	17	0.8121892962974020
2	0.0032114611466596	18	0.8683559394406505
3	0.0081009893216786	19	0.9125649996381605
4	0.0171240286619181	20	0.9453396205809574
5	0.0320720743527833	21	0.9680864942677585
6	0.0548307856028528	22	0.9827581789763112
7	0.0871361822564870	23	0.9914756203467121
8	0.1302923415174603	24	0.9961964092194694
9	0.1848955425508276	25	0.9984956609571091
10	0.2506163195331889	26	0.9994855586984285
11	0.3260874142923209	27	0.9998533730714648
12	0.4089316830907141	28	0.9999671864476404
13	0.4959414909423747	29	0.9999948432453556
14	0.5833939894958904	30	0.9999995655238333
15	0.6674601983218376	31	0.9999999961638728

Table 4.A.14 – Kaiser-Bessel window for SSR object type for other window sequences.

i	*w(i)*	*i*	*w(I)*
0	0.0005851230124487	128	0.7110428359000029
1	0.0009642149851497	129	0.7188474364707993

2	0.0013558207534965	130	0.7265597347077880
3	0.0017771849644394	131	0.7341770687621900
4	0.0022352533849672	132	0.7416968783634273
5	0.0027342299070304	133	0.7491167073477523
6	0.0032773001022195	134	0.7564342060337386
7	0.0038671998069216	135	0.7636471334404891
8	0.0045064443384152	136	0.7707533593446514
9	0.0051974336885144	137	0.7777508661725849
10	0.0059425050016407	138	0.7846377507242818
11	0.0067439602523141	139	0.7914122257259034
12	0.0076040812644888	140	0.7980726212080798
13	0.0085251378135895	141	0.8046173857073919
14	0.0095093917383048	142	0.8110450872887550
15	0.0105590986429280	143	0.8173544143867162
16	0.0116765080854300	144	0.8235441764639875
17	0.0128638627792770	145	0.8296133044858474
18	0.0141233971318631	146	0.8355608512093652
19	0.0154573353235409	147	0.8413859912867303
20	0.0168678890600951	148	0.8470880211822968
21	0.0183572550877256	149	0.8526663589032990
22	0.0199276125319803	150	0.8581205435445334
23	0.0215811201042484	151	0.8634502346476508
24	0.0233199132076965	152	0.8686552113760616
25	0.0251461009666641	153	0.8737353715068081
26	0.0270617631981826	154	0.8786907302411250
27	0.0290689473405856	155	0.8835214188357692
28	0.0311696653515848	156	0.8882276830575707
29	0.0333658905863535	157	0.8928098814640207
30	0.0356595546648444	158	0.8972684835130879
31	0.0380525443366107	159	0.9016040675058185
32	0.0405466983507029	160	0.9058173183656508
33	0.0431438043376910	161	0.9099090252587376
34	0.0458455957104702	162	0.9138800790599416
35	0.0486537485902075	163	0.9177314696695282
36	0.0515698787635492	164	0.9214642831859411
37	0.0545955386770205	165	0.9250796989403991
38	0.0577322144743916	166	0.9285789863994010
39	0.0609813230826460	167	0.9319635019415643
40	0.0643442093520723	168	0.9352346855155568
41	0.0678221432558827	169	0.9383940571861993
42	0.0714163171546603	170	0.9414432135761304
43	0.0751278431308314	171	0.9443838242107182
44	0.0789577503982528	172	0.9472176277741918
45	0.0829069827918993	173	0.9499464282852282
46	0.0869763963425241	174	0.9525720912004834
47	0.0911667569410503	175	0.9550965394547873
48	0.0954787380973307	176	0.9575217494469370
49	0.0999129187977865	177	0.9598497469802043
50	0.1044697814663005	178	0.9620826031668507
51	0.1091497100326053	179	0.9642224303060783
52	0.1139529881122542	180	0.9662713777449607
53	0.1188797973021148	181	0.9682316277319895
54	0.1239302155951605	182	0.9701053912729269
55	0.1291042159181728	183	0.9718949039986892
56	0.1344016647957880	184	0.9736024220549734
57	0.1398223211441467	185	0.9752302180233160
58	0.1453658351972151	186	0.9767805768831932

59	0.1510317475686540	187	0.9782557920246753
60	0.1568194884519144	188	0.9796581613210076
61	0.1627283769610327	189	0.9809899832703159
62	0.1687576206143887	190	0.9822535532154261
63	0.1749063149634756	191	0.9834511596505429
64	0.1811734433685097	192	0.9845850806232530
65	0.1875578769224857	193	0.9856575802399989
66	0.1940583745250518	194	0.9866709052828243
67	0.2006735831073503	195	0.9876272819448033
68	0.2074020380087318	196	0.9885289126911557
69	0.2142421635060113	197	0.9893779732525968
70	0.2211922734956977	198	0.9901766097569984
71	0.2282505723293797	199	0.9909269360049311
72	0.2354151558022098	200	0.9916310308941294
73	0.2426840122941792	201	0.9922909359973702
74	0.2500550240636293	202	0.9929086532976777
75	0.2575259686921987	203	0.9934861430841844
76	0.2650945206801527	204	0.9940253220113651
77	0.2727582531907993	205	0.9945280613237534
78	0.2805146399424422	206	0.9949961852476154
79	0.2883610572460804	207	0.9954314695504363
80	0.2962947861868143	208	0.9958356402684387
81	0.3043130149466800	209	0.9962103726017252
82	0.3124128412663888	210	0.9965572899760172
83	0.3205912750432127	211	0.9968779632693499
84	0.3288452410620226	212	0.9971739102014799
85	0.3371715818562547	213	0.9974465948831872
86	0.3455670606953511	214	0.9976974275220812
87	0.3540283646950029	215	0.9979277642809907
88	0.3625521080463003	216	0.9981389072844972
89	0.3711348353596863	217	0.9983321047686901
90	0.3797730251194006	218	0.9985085513687731
91	0.3884630932439016	219	0.9986693885387259
92	0.3972013967475546	220	0.9988157050968516
93	0.4059842374986933	221	0.9989485378906924
94	0.4148078660689724	222	0.9990688725744943
95	0.4236684856687616	223	0.9991776444921379
96	0.4325622561631607	224	0.9992757396582338
97	0.4414852981630577	225	0.9993639958299003
98	0.4504336971855032	226	0.9994432036616085
99	0.4594035078775303	227	0.9995141079353859
100	0.4683907582974173	228	0.9995774088586188
101	0.4773914542472655	229	0.9996337634216871
102	0.4864015836506502	230	0.9996837868076957
103	0.4954171209689973	231	0.9997280538466377
104	0.5044340316502417	232	0.9997671005064359
105	0.5134482766032377	233	0.9998014254134544
106	0.5224558166913167	234	0.9998314913952471
107	0.5314526172383208	235	0.9998577270385304
108	0.5404346525403849	236	0.9998805282555989
109	0.5493979103766972	237	0.9999002598526793
110	0.5583383965124314	238	0.9999172570940037
111	0.5672521391870222	239	0.9999318272557038
112	0.5761351935809411	240	0.9999442511639580
113	0.5849836462541291	241	0.9999547847121726
114	0.5937936195492526	242	0.9999636603523446
115	0.6025612759529649	243	0.9999710885561258

116	0.6112828224083939	244	0.9999772592414866
117	0.6199545145721097	245	0.9999823431612708
118	0.6285726610088878	246	0.9999864932503106
119	0.6371336273176413	247	0.9999898459281599
120	0.6456338401819751	248	0.9999925223548691
121	0.6540697913388968	249	0.9999946296375997
122	0.6624380414593221	250	0.9999962619864214
123	0.6707352239341151	251	0.9999975018180320
124	0.6789580485595255	252	0.9999984208055542
125	0.6871033051160131	253	0.9999990808746198
126	0.6951678668345944	254	0.9999995351446231
127	0.7031486937449871	255	0.9999998288155155

4.A.3 Differential scalefactor to index tables

Table 4.A.15 – transition table 0 (differential scalefactor to index)

DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX
0	68	16	87	32	46	48	25	64	9	80	40	96	96	112	112
1	69	17	88	33	47	49	19	65	10	81	43	97	97	113	113
2	70	18	89	34	48	50	20	66	12	82	44	98	98	114	114
3	71	19	72	35	49	51	14	67	13	83	45	99	99	115	115
4	75	20	90	36	50	52	15	68	17	84	52	100	100	116	116
5	76	21	73	37	51	53	16	69	18	85	53	101	101	117	117
6	77	22	65	38	41	54	11	70	21	86	63	102	102	118	118
7	78	23	66	39	42	55	7	71	22	87	56	103	103	119	119
8	79	24	58	40	35	56	8	72	26	88	64	104	104	120	120
9	80	25	67	41	36	57	5	73	27	89	57	105	105	121	121
10	81	26	59	42	37	58	2	74	28	90	74	106	106	122	122
11	82	27	60	43	29	59	1	75	31	91	91	107	107	123	123
12	83	28	61	44	38	60	0	76	32	92	92	108	108	124	124
13	84	29	62	45	30	61	3	77	33	93	93	109	109	125	125
14	85	30	54	46	23	62	4	78	34	94	94	110	110	126	126
15	86	31	55	47	24	63	6	79	39	95	95	111	111	127	127

Table 4.A.16 – transition table 1 (index to differential scalefactor)

INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF
0	60	16	53	32	76	48	34	64	88	80	9	96	96	112	112
1	59	17	68	33	77	49	35	65	22	81	10	97	97	113	113
2	58	18	69	34	78	50	36	66	23	82	11	98	98	114	114
3	61	19	49	35	40	51	37	67	25	83	12	99	99	115	115
4	62	20	50	36	41	52	84	68	0	84	13	100	100	116	116
5	57	21	70	37	42	53	85	69	1	85	14	101	101	117	117
6	63	22	71	38	44	54	30	70	2	86	15	102	102	118	118
7	55	23	46	39	79	55	31	71	3	87	16	103	103	119	119
8	56	24	47	40	80	56	87	72	19	88	17	104	104	120	120
9	64	25	48	41	38	57	89	73	21	89	18	105	105	121	121
10	65	26	72	42	39	58	24	74	90	90	20	106	106	122	122
11	54	27	73	43	81	59	26	75	4	91	91	107	107	123	123
12	66	28	74	44	82	60	27	76	5	92	92	108	108	124	124
13	67	29	43	45	83	61	28	77	6	93	93	109	109	125	125
14	51	30	45	46	32	62	29	78	7	94	94	110	110	126	126
15	52	31	75	47	33	63	86	79	8	95	95	111	111	127	127

4.A.4 Tables for TwinVQ

Table 4.A.17 – LSP codebook for core coder

filename	contents	mode	number of elements	number of vectors
20b19s48bs	LSP	core	20	64+16+2

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0.1174	0.2629	0.4047	0.5952	0.7249	0.8803	1.0465	1.1687	1.3174	1.4778	1.6125	1.779
	12	1.8951	2.0787	2.2225	2.3546	2.5002	2.6572	2.8355	2.9783				
1	0	0.1828	0.3386	0.4116	0.5621	0.721	0.8888	1.0088	1.1501	1.3288	1.4473	1.5727	1.7789
	12	1.8824	2.0304	2.1998	2.3585	2.4955	2.6511	2.814	2.9605				
2	0	0.1293	0.3249	0.4623	0.6582	0.7948	0.9477	1.1068	1.2104	1.3098	1.4293	1.569	1.757
	12	1.8739	2.0543	2.2133	2.363	2.5169	2.6593	2.8147	2.9676				
3	0	0.1359	0.3157	0.477	0.634	0.6863	0.797	1.0202	1.2062	1.3237	1.4243	1.571	1.788
	12	1.9209	2.0588	2.1726	2.3476	2.5309	2.6718	2.8079	2.9564				
4	0	0.1311	0.2912	0.4227	0.6131	0.7523	0.9053	1.0771	1.2221	1.3648	1.534	1.6759	1.8169
	12	1.902	2.0612	2.2039	2.3483	2.4994	2.6446	2.803	2.9602				
5	0	0.1264	0.2699	0.3763	0.581	0.7562	0.9114	1.0854	1.2163	1.373	1.5429	1.6488	1.7651
	12	1.8514	2.0338	2.1947	2.3444	2.4975	2.6497	2.8091	2.9622				
6	0	0.1437	0.2588	0.3554	0.5655	0.7454	0.9205	1.0799	1.171	1.3025	1.446	1.5693	1.7406
	12	1.8593	2.0314	2.1916	2.3495	2.5033	2.648	2.8094	2.9634				
7	0	0.1114	0.2943	0.4312	0.5984	0.7299	0.8952	1.0942	1.2548	1.3628	1.4482	1.5589	1.7495
	12	1.8897	2.0738	2.2136	2.3605	2.5199	2.6703	2.8201	2.9603				
8	0	0.0948	0.2009	0.3228	0.52	0.6759	0.8336	1.0034	1.169	1.3346	1.4869	1.6169	1.7833
	12	1.8962	2.0733	2.2408	2.3867	2.5308	2.6702	2.822	2.9715				
9	0	0.1354	0.2516	0.3765	0.5569	0.6437	0.7706	0.9587	1.1167	1.2847	1.448	1.5884	1.7576
	12	1.8759	2.0557	2.2131	2.3606	2.5117	2.6561	2.8135	2.9681				
10	0	0.1457	0.3447	0.4823	0.6299	0.7071	0.8266	1.0111	1.159	1.3168	1.4711	1.6095	1.7785
	12	1.8965	2.0814	2.2277	2.3749	2.5471	2.6981	2.8343	2.9631				
11	0	0.1308	0.2559	0.3849	0.5877	0.6716	0.7929	0.9911	1.1467	1.2875	1.4331	1.5804	1.7507
	12	1.8768	2.0533	2.2073	2.3571	2.5075	2.6526	2.8114	2.9668				
12	0	0.11	0.2342	0.3655	0.5678	0.7011	0.8539	1.0156	1.1344	1.2878	1.4475	1.5857	1.7553
	12	1.8788	2.0568	2.209	2.3575	2.5091	2.6533	2.8127	2.9664				
13	0	0.1276	0.2569	0.3779	0.5755	0.7111	0.8433	1.012	1.1497	1.2934	1.4561	1.5821	1.7393
	12	1.8597	2.0338	2.1981	2.3514	2.5005	2.6466	2.8099	2.9643				
14	0	0.1449	0.2648	0.3846	0.5961	0.7299	0.8612	1.0124	1.1361	1.276	1.4033	1.5325	1.7194
	12	1.8567	2.038	2.1982	2.3487	2.5037	2.6505	2.8093	2.9663				
15	0	0.1191	0.2724	0.3804	0.562	0.7344	0.8313	1.0047	1.1589	1.2462	1.4182	1.5826	1.7253
	12	1.8671	2.0428	2.1839	2.3524	2.4969	2.6468	2.8092	2.9619				
16	0	0.1605	0.3428	0.4476	0.5915	0.7067	0.8606	1.0336	1.1538	1.3018	1.4734	1.6169	1.7796
	12	1.9074	2.1093	2.2537	2.3646	2.5005	2.6536	2.8113	2.9592				
17	0	0.198	0.3043	0.3485	0.5137	0.6684	0.836	1.034	1.18	1.3348	1.4823	1.6125	1.7743
	12	1.8685	2.0274	2.1877	2.3437	2.5005	2.6494	2.8088	2.966				
18	0	0.1522	0.2816	0.3968	0.5617	0.7004	0.8729	1.0133	1.1174	1.2474	1.4405	1.6196	1.7822
	12	1.8452	2.0062	2.1803	2.3318	2.484	2.6414	2.8079	2.9638				
19	0	0.0838	0.2169	0.3715	0.6054	0.7642	0.8653	0.9821	1.1144	1.2896	1.4547	1.6024	1.7732
	12	1.8876	2.0665	2.2032	2.3498	2.507	2.6526	2.8113	2.9673				
20	0	0.175	0.3555	0.4484	0.5662	0.6469	0.7965	1.0043	1.1502	1.302	1.4691	1.6097	1.774
	12	1.8883	2.0712	2.2111	2.3544	2.5066	2.6538	2.813	2.9669				
21	0	0.1553	0.2518	0.3129	0.494	0.6681	0.84	1.0047	1.1351	1.2916	1.4545	1.6012	1.7689
	12	1.8857	2.0635	2.2054	2.3529	2.5061	2.6522	2.8115	2.9686				
22	0	0.151	0.2829	0.373	0.5356	0.672	0.8278	1.0133	1.1413	1.2788	1.4408	1.5875	1.7644
	12	1.8764	2.0482	2.1994	2.3476	2.5006	2.6488	2.8092	2.9658				
23	0	0.1242	0.2415	0.3427	0.5296	0.6996	0.8341	0.9758	1.1223	1.2826	1.4479	1.5923	1.7595
	12	1.8802	2.0585	2.1987	2.3476	2.5022	2.6495	2.8098	2.9672				
24	0	0.137	0.2369	0.3319	0.4979	0.6176	0.7802	0.9803	1.1446	1.3162	1.4684	1.6044	1.7707
	12	1.8874	2.063	2.2188	2.3651	2.5155	2.6591	2.8162	2.9694				

25	0	0.1082	0.2604	0.3933	0.5205	0.6302	0.8223	1.0015	1.1183	1.2792	1.4528	1.5918	1.7571
	12	1.8807	2.0589	2.1995	2.3482	2.5029	2.6504	2.8099	2.967				
26	0	0.1655	0.3502	0.4789	0.6442	0.7489	0.8833	1.0259	1.1458	1.276	1.4114	1.5562	1.7398
	12	1.8638	2.0451	2.2086	2.3589	2.5122	2.6577	2.8144	2.9662				
27	0	0.1398	0.2609	0.3662	0.539	0.6898	0.8598	1.0172	1.1053	1.2365	1.4228	1.5771	1.7457
	12	1.8741	2.0544	2.2075	2.3562	2.5101	2.6536	2.8122	2.9669				
28	0	0.0999	0.2563	0.3139	0.4686	0.6594	0.8453	1.0207	1.155	1.2962	1.4475	1.5626	1.7421
	12	1.8998	2.0971	2.2265	2.3679	2.5087	2.6455	2.7782	2.9457				
29	0	0.1138	0.2129	0.3229	0.5337	0.6762	0.8288	1.0247	1.1512	1.2682	1.4174	1.5696	1.7451
	12	1.8698	2.0503	2.2105	2.3574	2.5101	2.6545	2.8123	2.9678				
30	0	0.1464	0.2629	0.3682	0.5432	0.6492	0.8349	1.066	1.1853	1.2746	1.4003	1.5387	1.7244
	12	1.8658	2.0481	2.2015	2.3521	2.5052	2.6515	2.8106	2.967				
31	0	0.1281	0.2253	0.3181	0.5448	0.738	0.8919	1.0314	1.1421	1.2714	1.4252	1.5776	1.7548
	12	1.8818	2.06	2.2127	2.3616	2.5133	2.6548	2.8141	2.9675				
32	0	0.1671	0.2982	0.3779	0.5418	0.6953	0.8556	1.0118	1.1498	1.3094	1.4649	1.6072	1.7726
	12	1.8847	2.0659	2.2223	2.3653	2.5163	2.6591	2.8135	2.9657				
33	0	0.1584	0.313	0.4545	0.612	0.7032	0.8315	1.0019	1.0994	1.2239	1.4114	1.5721	1.7441
	12	1.8778	2.0618	2.1931	2.3407	2.4986	2.6482	2.8099	2.9659				
34	0	0.1343	0.3271	0.4606	0.6457	0.8085	0.9335	1.0208	1.1017	1.2672	1.4591	1.6018	1.7899
	12	1.9118	2.0595	2.2023	2.3582	2.512	2.6632	2.8272	2.9739				
35	0	0.1772	0.3363	0.4214	0.5756	0.727	0.8934	1.0098	1.0963	1.2554	1.4493	1.5827	1.7606
	12	1.892	2.0618	2.1981	2.3514	2.5063	2.6505	2.8097	2.9658				
36	0	0.1629	0.3196	0.422	0.5608	0.6796	0.8588	1.007	1.1124	1.3047	1.4725	1.5731	1.7454
	12	1.8934	2.0498	2.1861	2.3488	2.5041	2.6429	2.8095	2.9683				
37	0	0.196	0.3317	0.4323	0.6092	0.6915	0.8002	0.9693	1.1223	1.2892	1.4404	1.5735	1.7424
	12	1.8701	2.0495	2.201	2.3498	2.5061	2.6522	2.8101	2.9654				
38	0	0.1468	0.3363	0.4748	0.6645	0.7704	0.8472	0.9688	1.1161	1.2929	1.4648	1.6101	1.7753
	12	1.8918	2.0788	2.2212	2.3697	2.5156	2.6535	2.8097	2.9669				
39	0	0.2093	0.3679	0.4453	0.613	0.7225	0.8482	1.0341	1.1537	1.2889	1.4605	1.5785	1.7303
	12	1.8466	2.0314	2.1931	2.3403	2.4949	2.6475	2.8109	2.9672				
40	0	0.099	0.2279	0.3506	0.5357	0.6608	0.8117	0.993	1.1395	1.3025	1.4607	1.5989	1.7658
	12	1.883	2.062	2.2171	2.3647	2.5148	2.6583	2.8151	2.9689				
41	0	0.1526	0.2818	0.3926	0.5601	0.6776	0.7982	0.9554	1.0972	1.2681	1.4457	1.5934	1.7644
	12	1.8867	2.0678	2.2277	2.3753	2.5217	2.661	2.8151	2.9671				
42	0	0.1664	0.2865	0.3753	0.5542	0.7075	0.8239	0.9482	1.1018	1.2804	1.4409	1.5888	1.7569
	12	1.876	2.0564	2.2125	2.3592	2.5109	2.6547	2.8128	2.9673				
43	0	0.1497	0.2776	0.404	0.6093	0.6903	0.8241	0.9759	1.0756	1.2293	1.4062	1.564	1.7444
	12	1.869	2.0471	2.2058	2.3548	2.5082	2.6541	2.8123	2.9663				
44	0	0.1856	0.3246	0.416	0.5572	0.6681	0.8443	1.0393	1.1434	1.2551	1.4226	1.5953	1.7619
	12	1.8762	2.0626	2.2266	2.3616	2.5056	2.6552	2.8174	2.9657				
45	0	0.1963	0.3169	0.3824	0.5766	0.6988	0.8159	1.013	1.111	1.2497	1.4341	1.5718	1.7518
	12	1.8737	2.051	2.2107	2.3543	2.5119	2.6529	2.812	2.9667				
46	0	0.1856	0.3454	0.4117	0.5668	0.7447	0.8623	0.9852	1.169	1.2915	1.3994	1.5688	1.7447
	12	1.8442	2.0565	2.2031	2.3417	2.5085	2.6514	2.8045	2.9678				
47	0	0.166	0.2923	0.4013	0.5757	0.6739	0.8321	1.0109	1.1455	1.2546	1.3844	1.5564	1.7446
	12	1.8702	2.0453	2.2067	2.3576	2.5086	2.6522	2.8103	2.9659				
48	0	0.1399	0.3315	0.4603	0.6454	0.7743	0.9153	1.0794	1.2339	1.3972	1.5176	1.5984	1.7353
	12	1.8565	2.0569	2.2083	2.3508	2.5053	2.6524	2.8188	2.9722				
49	0	0.1288	0.3059	0.4501	0.6671	0.7666	0.8348	0.9684	1.1432	1.3353	1.5161	1.6186	1.7363
	12	1.8551	2.0594	2.207	2.3614	2.5017	2.6432	2.8081	2.9672				
50	0	0.1283	0.3108	0.4497	0.6447	0.7964	0.8913	0.9786	1.1139	1.3089	1.4954	1.6582	1.7989
	12	1.8601	2.0313	2.2017	2.357	2.5284	2.6688	2.8041	2.953				
51	0	0.1353	0.3106	0.4598	0.6512	0.7234	0.8098	0.9998	1.1857	1.3597	1.486	1.5771	1.7548
	12	1.9111	2.1113	2.2261	2.3281	2.4854	2.6665	2.8452	2.9846				
52	0	0.1604	0.3143	0.4318	0.6003	0.7132	0.8728	1.0448	1.1722	1.3083	1.4632	1.5988	1.7629
	12	1.8779	2.0539	2.2067	2.3572	2.5069	2.6503	2.8111	2.9627				
53	0	0.1028	0.2826	0.45	0.6391	0.7665	0.9555	1.1078	1.1585	1.2412	1.4131	1.5947	1.7693

	12	1.8957	2.0951	2.2557	2.3628	2.503	2.6596	2.8153	2.9674				
54	0	0.1349	0.3392	0.4855	0.6061	0.6773	0.8411	1.0665	1.1875	1.2882	1.4453	1.6329	1.8118
	12	1.8896	2.035	2.201	2.3721	2.5096	2.6424	2.8095	2.9693				
55	0	0.2062	0.3369	0.3912	0.5671	0.7037	0.8188	1.0099	1.1641	1.3288	1.4701	1.5703	1.7315
	12	1.8461	2.022	2.1849	2.3378	2.497	2.6473	2.8072	2.9652				
56	0	0.1611	0.2965	0.3994	0.576	0.7066	0.8527	1.0117	1.1601	1.3263	1.4822	1.6103	1.7697
	12	1.885	2.037	2.1847	2.3501	2.5087	2.6489	2.8095	2.963				
57	0	0.174	0.3077	0.3979	0.5763	0.6674	0.7658	0.9572	1.1408	1.3099	1.4713	1.611	1.7712
	12	1.8662	2.0278	2.1799	2.3338	2.4943	2.6458	2.8068	2.9654				
58	0	0.1644	0.2687	0.3549	0.558	0.7462	0.8957	0.9979	1.1157	1.2975	1.4555	1.5783	1.722
	12	1.8323	2.0177	2.1885	2.3429	2.5006	2.6499	2.8095	2.9659				
59	0	0.1396	0.3211	0.4387	0.591	0.7217	0.8955	1.059	1.1835	1.346	1.5024	1.6307	1.8283
	12	1.9458	2.0646	2.1776	2.3532	2.5179	2.6556	2.8191	2.9694				
60	0	0.182	0.288	0.3636	0.5345	0.6327	0.7978	1.0031	1.1387	1.2978	1.4558	1.5979	1.7627
	12	1.8847	2.061	2.1793	2.354	2.5294	2.6627	2.8095	2.9644				
61	0	0.1252	0.2473	0.3853	0.5624	0.7105	0.8498	0.9762	1.1477	1.309	1.4242	1.5993	1.7636
	12	1.8327	2.0352	2.2026	2.332	2.5015	2.6514	2.8022	2.9628				
62	0	0.2073	0.3438	0.3918	0.5472	0.7194	0.8508	1.0187	1.176	1.2857	1.4605	1.6015	1.7364
	12	1.8728	2.0492	2.1888	2.3526	2.5016	2.6466	2.8089	2.9626				
63	0	0.1434	0.2665	0.369	0.5565	0.6735	0.8049	1.0074	1.1739	1.3292	1.4451	1.5527	1.7176
	12	1.8464	2.0366	2.1979	2.3497	2.5038	2.6518	2.8105	2.9653				
64	0	0.0561	0.0312	0.0155	-0.0042	0.0325	0.0171	0.0047	0.0264	0.0483	0.0216	-0.0111	-0.0329
	12	0.0229	0.0045	-0.0032	0.0015	0.0081	0.0198	0.0181	0.0193				
65	0	0.0423	0.0088	-0.0058	-0.0414	-0.0039	0.0202	0.0252	0.0543	0.0252	-0.0252	-0.009	-0.003
	12	0.0404	0.0114	0.068	0.1005	0.0661	0.0072	-0.0267	-0.0069				
66	0	-0.0397	-0.0683	-0.0203	-0.0272	0.0078	0.004	-0.0156	0.0067	0.0163	0.0187	0.0435	0.0378
	12	0.0811	0.0438	-0.0306	-0.0252	-0.0009	0.0161	0.0134	0.0158				
67	0	0.02	-0.0239	-0.0162	-0.01	0.0278	0.0185	0.0115	0.0347	0.0359	0.024	0.0296	0.0116
	12	0.0473	0.0197	0.0079	0.0177	0.0266	0.043	0.0641	0.0436				
68	0	-0.0128	0.0234	0.0498	-0.0345	-0.0268	0.0514	0.0364	0.0499	0.0485	0.0428	0.0564	0.0481
	12	0.0919	0.0574	0.013	0.0282	0.0017	-0.0071	0.0073	0.0156				
69	0	-0.0074	0.0132	0.0242	-0.0196	0.0217	0.0452	0.0326	0.0172	0.0195	0.0434	0.0361	-0.0285
	12	-0.0001	0.0035	0.0074	-0.0221	-0.0314	-0.0046	0.0049	0.0143				
70	0	-0.0005	-0.0231	0	-0.0433	-0.019	-0.0067	-0.0218	0.0094	0.0235	0.0337	0.0322	-0.0196
	12	0.0049	-0.0037	0.0272	-0.0061	-0.0058	0.021	0.017	0.0165				
71	0	0.0121	-0.0081	0.0233	0.0268	0.0422	-0.0014	-0.0065	0.0172	0.013	0.0054	0.0201	0.0062
	12	0.0431	0.016	0.0193	0.0402	0.0449	0.0152	-0.0066	0.0082				
72	0	-0.0388	-0.0291	0.0236	-0.0205	-0.0011	-0.0001	-0.0203	0.0054	0.0064	-0.0207	-0.0137	-0.0211
	12	0.0169	-0.0032	-0.0093	-0.0004	0.0357	0.0354	-0.0014	-0.0013				
73	0	-0.0215	-0.0158	0.0584	0.0477	0.0426	0.0422	0.0241	0.0117	-0.0025	-0.0027	0.0182	0.0029
	12	0.0412	0.0151	0.0549	0.0632	0.0653	0.0757	0.0695	0.0421				
74	0	0.0022	0.0325	0.0862	0.0287	0.0114	0.013	0.0068	-0.0033	-0.0203	-0.0141	0.0073	-0.0058
	12	0.0333	0.0088	0.0676	0.0589	0.0051	-0.0155	-0.0006	0.0127				
75	0	-0.0292	-0.0195	0.0575	0.0156	0.0006	0.0107	-0.0123	0.0147	0.0253	0.0278	0.0644	0.0385
	12	0.0303	-0.0152	-0.0222	0.0136	0.0206	0.0226	0.0215	0.0208				
76	0	-0.0255	-0.0425	0.0049	0.0131	0.0372	0.039	-0.015	-0.0178	0.0075	0.0045	0.0157	0.002
	12	0.0357	0.0057	0.0122	0.0176	0.0198	0.0277	0.0226	0.0208				
77	0	0.0567	0.052	0.0237	-0.0588	-0.0326	-0.0352	-0.0394	0.0085	0.0105	0.0036	0.0152	0.002
	12	0.0395	0.0105	0.028	0.0407	0.0652	0.0661	0.0186	-0.008				
78	0	0.032	0.0601	0.0632	-0.0162	-0.0058	0	-0.0441	-0.0256	-0.0027	-0.0019	0.017	0.0029
	12	0.0321	0.0081	0.0478	0.026	0.0004	0.0197	0.0281	0.0243				
79	0	0.0268	0.0004	0.0397	0.0076	-0.0044	0.0231	0.0238	0.0318	-0.0023	-0.0106	0.0283	0.0012
	12	0.0055	-0.0158	0.0285	0.032	0.0307	0.0348	0.0273	0.0219				
80	0	0.4987	0.4924	0.4713	0.4427	0.4144	0.4168	0.3813	0.3695	0.3672	0.3619	0.3276	0.3173
	12	0.3115	0.278	0.3223	0.2951	0.2826	0.2568	0.2145	0.2527				
81	0	0.3519	0.3181	0.2607	0.206	0.1473	0.1563	0.1073	0.0633	0.0178	0.0178	-0.03	-0.0538
	12	-0.0466	-0.0529	0.0389	-0.024	-0.0405	-0.1279	-0.1221	0.0012				

Table 4.A.18 – LSP codebook for scalable coder

filename	contents	mode	number of elements	number of vectors
20b19s48sc	LSP	enhance	20	64+16+2

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0.1567	0.293	0.4518	0.6063	0.7601	0.9216	1.067	1.2147	1.3665	1.5191	1.6681	1.8001
	12	1.9483	2.1034	2.2565	2.3958	2.5371	2.6784	2.8201	2.9648				
1	0	0.1363	0.2451	0.3925	0.524	0.6463	0.8015	0.9244	1.0729	1.2474	1.4157	1.5815	1.7372
	12	1.8908	2.0462	2.2112	2.3506	2.4974	2.653	2.7995	2.9567				
2	0	0.1649	0.2806	0.4123	0.5608	0.7206	0.8699	0.9973	1.1528	1.3042	1.45	1.6105	1.7487
	12	1.8897	2.0482	2.2069	2.3509	2.499	2.6508	2.7986	2.9581				
3	0	0.1559	0.2632	0.4053	0.5361	0.7076	0.8445	0.9292	1.0251	1.1718	1.3625	1.5449	1.7128
	12	1.879	2.0404	2.2004	2.3505	2.5068	2.6627	2.82	2.9759				
4	0	0.1529	0.2209	0.3506	0.4867	0.6205	0.7836	0.8989	1.0645	1.2688	1.3758	1.4801	1.6433
	12	1.8303	2	2.1712	2.3409	2.5026	2.6545	2.817	2.9744				
5	0	0.1433	0.2288	0.3578	0.4937	0.6217	0.7847	0.9273	1.0626	1.2336	1.3957	1.5397	1.704
	12	1.8573	2.0034	2.1756	2.3409	2.5	2.6518	2.807	2.9664				
6	0	0.1664	0.2349	0.3465	0.4486	0.5634	0.7389	0.9071	1.0751	1.243	1.3872	1.5295	1.7175
	12	1.8303	1.9659	2.1052	2.2759	2.4874	2.6327	2.7668	2.9633				
7	0	0.1458	0.2419	0.3502	0.4939	0.6698	0.8162	0.9193	1.0387	1.2036	1.3958	1.5755	1.7351
	12	1.8583	1.9894	2.163	2.3414	2.4937	2.6384	2.7951	2.9633				
8	0	0.1491	0.2608	0.4017	0.5588	0.719	0.8763	1.0197	1.1674	1.3192	1.477	1.6349	1.7749
	12	1.9286	2.0907	2.2376	2.3789	2.5253	2.6725	2.8259	2.975				
9	0	0.1812	0.2982	0.4199	0.549	0.6944	0.7986	0.8602	0.9987	1.2019	1.4017	1.5757	1.7307
	12	1.8879	2.05	2.1715	2.319	2.4852	2.6498	2.8074	2.9718				
10	0	0.164	0.2604	0.3885	0.5396	0.6878	0.8403	0.9959	1.1211	1.2562	1.4133	1.5712	1.7299
	12	1.8956	2.0557	2.2103	2.3572	2.5108	2.6634	2.8211	2.9763				
11	0	0.1566	0.297	0.4632	0.5803	0.6433	0.7481	0.9195	1.1048	1.2769	1.4439	1.6033	1.7564
	12	1.9081	2.0646	2.2041	2.349	2.4886	2.6269	2.7689	2.9472				
12	0	0.1467	0.2153	0.361	0.5554	0.7325	0.8634	0.9628	1.1097	1.3069	1.4554	1.5611	1.6825
	12	1.8505	2.0227	2.1789	2.3352	2.4982	2.6573	2.815	2.9725				
13	0	0.1285	0.2004	0.354	0.5039	0.6163	0.7581	0.9083	1.0782	1.2559	1.4238	1.5749	1.724
	12	1.8709	2.0279	2.1951	2.3786	2.541	2.6805	2.8318	2.9801				
14	0	0.1463	0.2438	0.3903	0.5417	0.6666	0.7941	0.9412	1.1156	1.295	1.4665	1.6099	1.722
	12	1.8562	2.0184	2.1976	2.3507	2.5054	2.6602	2.8182	2.9738				
15	0	0.1344	0.1615	0.2467	0.4187	0.6347	0.81	0.9542	1.1096	1.2665	1.4258	1.5829	1.7285
	12	1.8835	2.0436	2.1891	2.3415	2.4915	2.6198	2.7569	2.9449				
16	0	0.187	0.3036	0.4315	0.535	0.6405	0.7804	0.9596	1.1519	1.3195	1.4341	1.5404	1.6821
	12	1.8442	2.009	2.1585	2.3114	2.4752	2.6417	2.8005	2.9638				
17	0	0.1751	0.2719	0.3828	0.5296	0.7036	0.9034	1.0637	1.1863	1.3125	1.4169	1.5111	1.6425
	12	1.8131	1.9929	2.1476	2.3055	2.4739	2.6423	2.8053	2.9703				
18	0	0.1547	0.3035	0.4669	0.5746	0.672	0.8425	1.033	1.1782	1.2909	1.4494	1.6306	1.7793
	12	1.8982	2.038	2.2171	2.378	2.4998	2.6364	2.808	2.9789				
19	0	0.16	0.2923	0.4369	0.5654	0.6968	0.8422	0.9896	1.1608	1.3081	1.4465	1.5856	1.7046
	12	1.8411	1.9932	2.1526	2.3165	2.4797	2.6425	2.8061	2.968				
20	0	0.1562	0.2721	0.4059	0.5534	0.7148	0.8458	0.9977	1.1582	1.3127	1.5004	1.6237	1.7315
	12	1.912	2.0342	2.1649	2.3464	2.4693	2.6224	2.7995	2.9671				
21	0	0.0886	0.1482	0.3145	0.5203	0.7025	0.8746	1.0101	1.1515	1.3107	1.4647	1.6207	1.7669
	12	1.9143	2.0666	2.1859	2.3295	2.4918	2.6541	2.8129	2.9723				
22	0	0.1526	0.2884	0.4275	0.5815	0.7543	0.9188	1.0653	1.2176	1.3491	1.4997	1.69	1.8325
	12	1.9063	1.9897	2.1563	2.3462	2.4988	2.6451	2.7962	2.9644				
23	0	0.1488	0.2703	0.4265	0.5861	0.744	0.8934	1.0286	1.1766	1.3575	1.5352	1.646	1.6996
	12	1.8178	2.0112	2.2095	2.3733	2.4985	2.632	2.7964	2.9644				
24	0	0.1572	0.2429	0.3593	0.5062	0.6562	0.8095	0.961	1.118	1.2825	1.4421	1.6008	1.7497
	12	1.901	2.0565	2.208	2.3977	2.5566	2.6866	2.8352	2.9804				

25	0	0.1483	0.2405	0.3764	0.5123	0.6309	0.7726	0.9575	1.1598	1.2968	1.3939	1.5335	1.6989
	12	1.8671	2.0255	2.1687	2.3193	2.4833	2.6474	2.8075	2.9693				
26	0	0.14	0.2833	0.4514	0.588	0.7017	0.838	1.0121	1.1987	1.3348	1.4524	1.6014	1.769
	12	1.9371	2.0673	2.2007	2.3651	2.5405	2.6912	2.8146	2.9573				
27	0	0.147	0.2935	0.4763	0.6216	0.7255	0.8457	0.9767	1.1172	1.2633	1.4177	1.5811	1.744
	12	1.8958	2.0268	2.1556	2.3108	2.476	2.6424	2.8053	2.9671				
28	0	0.1452	0.2348	0.3705	0.5244	0.6837	0.8371	0.9823	1.1497	1.3086	1.4343	1.5756	1.734
	12	1.8896	2.043	2.1931	2.3458	2.5031	2.6589	2.818	2.9747				
29	0	0.1752	0.2676	0.3921	0.5289	0.6385	0.8015	1.0014	1.1711	1.3076	1.4502	1.5964	1.7216
	12	1.8779	2.0451	2.1952	2.3347	2.4832	2.6366	2.7888	2.9543				
30	0	0.1758	0.2815	0.3937	0.5605	0.6842	0.7915	0.9867	1.1111	1.2982	1.4349	1.5369	1.7311
	12	1.8721	2.0534	2.1813	2.31	2.4778	2.6298	2.7919	2.9531				
31	0	0.1484	0.2614	0.406	0.5621	0.6949	0.838	1.0034	1.1408	1.3059	1.4644	1.5745	1.7082
	12	1.8647	2.0263	2.1836	2.3269	2.4801	2.6403	2.7939	2.9558				
32	0	0.1859	0.2648	0.405	0.5544	0.7034	0.853	0.9523	1.0875	1.2349	1.3468	1.4844	1.6538
	12	1.8288	1.9979	2.1465	2.3027	2.4674	2.6434	2.8068	2.9675				
33	0	0.1523	0.2676	0.4172	0.5676	0.6782	0.7917	0.938	1.0911	1.2527	1.4017	1.5428	1.7005
	12	1.8617	2.0219	2.1783	2.3305	2.4898	2.6492	2.809	2.9696				
34	0	0.1429	0.2452	0.3811	0.5172	0.6436	0.78	0.9157	1.0533	1.2193	1.3916	1.5431	1.6877
	12	1.8483	2.0113	2.1709	2.3314	2.4933	2.6543	2.8159	2.9774				
35	0	0.1429	0.2755	0.4623	0.6348	0.7367	0.8006	0.9024	1.082	1.2703	1.4362	1.597	1.7439
	12	1.8992	2.0596	2.2391	2.3879	2.5199	2.6538	2.8044	2.9714				
36	0	0.1755	0.2674	0.3797	0.5093	0.6128	0.768	0.9463	1.0888	1.2435	1.3842	1.5028	1.653
	12	1.8138	1.9719	2.1481	2.3143	2.4863	2.643	2.8036	2.9727				
37	0	0.1028	0.2019	0.4564	0.5772	0.6168	0.7817	1.0249	1.1519	1.2362	1.3784	1.5467	1.7077
	12	1.8748	2.0347	2.1629	2.3126	2.4791	2.633	2.7958	2.9645				
38	0	0.1552	0.2539	0.3842	0.4973	0.6185	0.7823	0.9783	1.2079	1.3287	1.4312	1.5682	1.6595
	12	1.7912	1.9505	2.1305	2.3021	2.4812	2.6502	2.8077	2.973				
39	0	0.144	0.2506	0.4179	0.5561	0.6573	0.7961	0.9659	1.1313	1.2651	1.3966	1.5404	1.691
	12	1.85	2.0157	2.1777	2.334	2.4927	2.6505	2.8105	2.9723				
40	0	0.1231	0.2362	0.392	0.5375	0.6822	0.839	0.9828	1.1363	1.2963	1.4559	1.6114	1.7592
	12	1.9089	2.0622	2.2153	2.3634	2.5155	2.6669	2.8236	2.9771				
41	0	0.1644	0.3065	0.4254	0.5421	0.7221	0.8851	0.9748	1.1407	1.3215	1.4231	1.5704	1.757
	12	1.8728	2.0114	2.1897	2.3369	2.4832	2.6524	2.7991	2.9625				
42	0	0.1541	0.2444	0.3696	0.4892	0.584	0.7198	0.9029	1.0817	1.2504	1.4125	1.5675	1.7227
	12	1.8809	2.0445	2.2064	2.3566	2.5114	2.6657	2.8226	2.9783				
43	0	0.1685	0.2822	0.4073	0.531	0.678	0.826	0.9442	1.0943	1.2424	1.3927	1.5633	1.721
	12	1.8769	2.0336	2.1866	2.3397	2.4986	2.6565	2.8165	2.9738				
44	0	0.1595	0.2721	0.4006	0.518	0.6605	0.8306	0.9808	1.1277	1.3003	1.4401	1.5503	1.6986
	12	1.8766	2.0486	2.2281	2.3824	2.5148	2.6498	2.7995	2.9672				
45	0	0.1576	0.2128	0.3353	0.4837	0.5821	0.6951	0.8633	1.013	1.2138	1.4012	1.5417	1.7026
	12	1.8413	2.004	2.1891	2.342	2.5054	2.6629	2.8164	2.9719				
46	0	0.1623	0.2444	0.3882	0.541	0.6619	0.8141	0.9687	1.1125	1.2987	1.4352	1.5293	1.6206
	12	1.7733	1.9671	2.1671	2.3417	2.5032	2.6485	2.8044	2.975				
47	0	0.1645	0.2494	0.4065	0.5624	0.7081	0.8553	0.9783	1.117	1.2605	1.3666	1.5037	1.6825
	12	1.8595	2.0285	2.2063	2.3573	2.512	2.6642	2.8209	2.9768				
48	0	0.1412	0.2599	0.4089	0.5763	0.7575	0.9404	1.0963	1.2039	1.2875	1.4104	1.5831	1.7469
	12	1.9057	2.0655	2.2198	2.3592	2.5008	2.6519	2.8075	2.9645				
49	0	0.1506	0.2249	0.3354	0.4702	0.6394	0.8209	0.9874	1.1137	1.2158	1.3713	1.5632	1.7234
	12	1.8511	2.0006	2.1659	2.3298	2.4952	2.6518	2.8204	2.9774				
50	0	0.1427	0.2519	0.4087	0.5698	0.73	0.8871	1.0136	1.1402	1.2784	1.4188	1.5715	1.7279
	12	1.8821	2.0373	2.1915	2.3442	2.5019	2.6578	2.8168	2.9741				
51	0	0.1546	0.2464	0.3871	0.5358	0.6887	0.8628	1.0297	1.1593	1.2707	1.3877	1.5348	1.7029
	12	1.8544	2.0102	2.1646	2.3194	2.485	2.647	2.8099	2.971				
52	0	0.1563	0.2624	0.3977	0.5517	0.7151	0.8666	0.9931	1.1265	1.2623	1.419	1.5703	1.7052
	12	1.8638	2.0244	2.1786	2.333	2.4926	2.6513	2.8121	2.9714				
53	0	0.1756	0.3055	0.4336	0.5642	0.6999	0.8519	1.0053	1.1666	1.3314	1.4941	1.6493	1.7804

	12	1.9026	2.041	2.1964	2.3556	2.5057	2.6603	2.8194	2.972				
54	0	0.1584	0.3036	0.4517	0.5956	0.7302	0.8717	1.0275	1.1963	1.3597	1.5018	1.6108	1.7443
	12	1.9113	2.0817	2.223	2.3474	2.489	2.6508	2.8187	2.9778				
55	0	0.1863	0.2831	0.4032	0.5544	0.7006	0.8505	1.0197	1.1847	1.3216	1.427	1.5489	1.695
	12	1.8516	2.0187	2.152	2.3037	2.4735	2.6409	2.804	2.9682				
56	0	0.1365	0.2121	0.3269	0.4585	0.6069	0.7824	0.9466	1.1138	1.282	1.4431	1.6006	1.7496
	12	1.9023	2.0596	2.2231	2.3686	2.5198	2.6706	2.8259	2.9791				
57	0	0.1418	0.2012	0.3235	0.4318	0.5561	0.7305	0.9193	1.0594	1.2154	1.3873	1.5191	1.6691
	12	1.8355	2.0049	2.1635	2.3106	2.4786	2.6498	2.7995	2.9615				
58	0	0.1298	0.2535	0.4368	0.6313	0.8039	0.9111	0.9697	1.0732	1.2481	1.4316	1.5946	1.7482
	12	1.8986	2.0577	2.219	2.3698	2.5216	2.6719	2.829	2.9839				
59	0	0.164	0.2683	0.3859	0.5464	0.721	0.8791	0.9947	1.1248	1.2728	1.42	1.5777	1.6956
	12	1.8111	1.9719	2.1449	2.334	2.5168	2.6671	2.83	2.982				
60	0	0.16	0.2608	0.4067	0.5504	0.6802	0.817	0.9717	1.1255	1.2888	1.4648	1.6376	1.8126
	12	1.9239	2.0146	2.1353	2.3044	2.4523	2.618	2.7863	2.9651				
61	0	0.1642	0.2736	0.3797	0.5014	0.6611	0.8243	0.9655	1.1203	1.2834	1.4431	1.604	1.7534
	12	1.9015	2.0469	2.1684	2.318	2.484	2.6484	2.8086	2.9696				
62	0	0.15	0.273	0.4163	0.5685	0.7419	0.9107	1.0222	1.1298	1.2903	1.4624	1.6191	1.7762
	12	1.9303	2.0511	2.1871	2.3528	2.5078	2.6608	2.8287	2.9768				
63	0	0.1448	0.2603	0.3473	0.4684	0.6635	0.8312	0.9606	1.124	1.2812	1.3949	1.5717	1.7249
	12	1.9085	2.0455	2.1565	2.29	2.4086	2.6323	2.8375	2.9776				
64	0	0.0383	0.0567	0.0096	0.0058	0.0176	0.0025	0.0011	-0.0004	0.0199	0.0127	-0.0218	-0.0168
	12	0.0079	0.0142	-0.0017	0.0067	0.0106	0.0125	0.01	0.0094				
65	0	0.0081	0.0043	-0.0335	-0.0402	-0.012	0.0213	0.042	0.0507	0.0238	-0.0191	-0.0249	-0.0022
	12	0.0109	0.0117	0.0497	0.0557	0.0464	0.035	0.0275	0.019				
66	0	-0.0261	-0.048	-0.046	-0.014	0.0035	0.0071	0.0125	0.0034	-0.0005	0.0004	0.0096	0.034
	12	0.0524	0.0495	-0.0303	-0.0281	-0.008	0.0009	-0.0004	0.0039				
67	0	0.0102	-0.0001	-0.0296	-0.0006	0.0327	0.0346	0.0448	0.0414	0.0344	0.0273	0.0219	0.0264
	12	0.0257	0.0208	-0.012	-0.0134	0.0141	0.0377	0.0392	0.0162				
68	0	0.0104	0.0071	0.011	0.0152	-0.0102	0.0115	0.0289	0.0372	0.0476	0.0562	0.0607	0.0667
	12	0.0618	0.0449	0.0256	-0.0128	-0.0182	0.0032	0.005	0.0107				
69	0	-0.0177	0.0338	0.0173	0.0015	0.0224	0.0144	0.034	0.0447	0.0526	0.0358	-0.0029	-0.0166
	12	-0.0123	-0.0052	0.0066	-0.0246	-0.0336	-0.0143	-0.0047	0.0054				
70	0	-0.0163	0.0226	0.0214	-0.0236	-0.0324	-0.0239	-0.0003	0.0098	-0.0045	-0.0015	0.0045	0.0126
	12	0.0129	0.0104	-0.0208	-0.0085	0.0011	0.0066	0.0228	0.0289				
71	0	0.0096	0.0184	0.0142	0.0422	0.0419	-0.0037	0.015	0.0267	0.0238	0.0191	0.0134	0.0177
	12	0.0218	0.0207	0.041	0.0594	0.0599	0.0113	-0.0264	-0.0092				
72	0	-0.0428	-0.0164	-0.002	-0.0084	-0.0008	0.0143	0.0203	0.0009	-0.0134	-0.0132	-0.0215	-0.0289
	12	-0.0202	-0.0083	0.0009	0.0297	0.0171	-0.015	-0.0062	0.0079				
73	0	-0.0259	0.013	0.0487	0.0691	0.0466	0.035	0.0465	0.0052	-0.0155	0.0023	0.0134	0.0146
	12	0.0142	0.011	0.029	0.0326	0.0437	0.0669	0.0835	0.0498				
74	0	-0.0035	0.0321	0.054	0.0382	0.0055	0.0101	0.0146	-0.0031	-0.0324	-0.0381	-0.0189	0.0044
	12	0.0177	0.0185	0.0471	0.0252	-0.0066	-0.0072	-0.0129	-0.0052				
75	0	-0.0373	-0.0028	0.0375	0.0295	-0.0016	-0.0017	-0.0022	0.0084	0.0219	0.0322	0.0421	0.0369
	12	0.0094	-0.0019	0.0096	0.0242	0.0324	0.031	0.0363	0.0306				
76	0	-0.0433	-0.0333	0.001	0.0338	0.034	0.0181	-0.0051	-0.0337	-0.017	0.0115	0.0188	0.0132
	12	0.011	0.0114	0.0232	0.025	0.0232	0.0203	0.015	0.0128				
77	0	0.0552	0.0672	-0.0093	-0.0535	-0.022	-0.0204	-0.0284	-0.0194	-0.0177	-0.0228	-0.0217	-0.0114
	12	-0.0043	-0.0033	-0.0012	0.0158	0.0533	0.0514	0.0011	-0.0072				
78	0	0.0209	0.0653	0.0635	0.0541	0.0327	-0.0325	-0.0418	-0.0221	-0.002	0.0078	0.0096	0.0202
	12	0.0285	0.0278	0.0574	0.0428	-0.0096	-0.011	0.026	0.0358				
79	0	0.0146	0.062	0.0578	0.0008	-0.0285	0.0054	0.0183	0.0136	0.0081	0.001	-0.0056	0.0145
	12	-0.0033	-0.0204	0.038	0.0141	0.0198	0.0573	0.031	-0.0055				
80	0	0.4989	0.4993	0.499	0.4995	0.4999	0.4985	0.4973	0.499	0.4977	0.4971	0.4977	0.4995
	12	0.4969	0.4986	0.4996	0.4996	0.4999	0.4989	0.4937	0.4906				
81	0	0.2664	0.3126	0.3112	0.3132	0.2987	0.2609	0.2724	0.2535	0.2363	0.2679	0.2746	0.2536
	12	0.2708	0.269	0.2831	0.2939	0.262	0.2428	0.2456	0.2665				

Table 4.A.19 – Interleave VQ codebook 0 for core coder (WINDOW TYPE : LONG)

filename	Contents	mode	number of elements	number of vectors
cdmdct_0	MDCT	core LONG	20	32

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	-165.2	44.8	83.7	-31.8	-54.7	79.8	104.3	-11322.7	-46.7	-5.7	139.8	-38.4
	12	-59.7	-101.2	-5.6	134.4	34	113	-738.8	80.1				
1	0	17.2	4.2	148.2	54.7	54.2	-11.9	74.7	35.2	73.9	90	9523.9	73.2
	12	1.1	-47.8	-5.7	-966.6	60	120.8	-189.5	-59.4				
2	0	395.5	136	5816.1	-64.1	-89.2	17.3	371.2	-9.6	11.2	-10.6	-31.9	27.1
	12	-41.5	31.5	-496.7	-138.5	-2108.3	-272.2	-99.2	262.8				
3	0	-1136.4	-142.9	73.5	-92	-544.9	-28.6	-79.1	-85.1	136.3	-5.4	-122.4	-81.2
	12	-6.2	5.8	-123.2	11554.5	4.3	-71.2	-22.1	-203.1				
4	0	-56	14.4	30	-2.6	29.7	-150	96.5	71.3	24.7	-130.8	-7.1	5682.9
	12	-7.6	43.1	15.6	35.9	-219.7	-84.9	-1051.1	647.5				
5	0	209.2	-93.8	-105.8	-18.2	51.5	22.9	-38.3	88	4.6	-38.3	-118.2	328.6
	12	12156.2	-111	-47	109.5	-3.2	116.4	-156.3	51				
6	0	69.1	-74.3	15.4	184.4	-165.6	-97.8	23.8	12.2	-12330.6	-72.9	57.2	42
	12	-55.7	61.3	-107.8	98.3	167.9	32.8	146	6.3				
7	0	58.7	728	-3068.5	-263.4	-353.1	-398.7	2696.7	95.2	-202.9	-28.2	-5.6	45.7
	12	22.5	13.1	-66.7	42	-269.3	-467.2	-208.8	2168				
8	0	17.8	-3857.2	-214.1	221.1	-3488.2	83.3	-162.9	-260.7	3.5	-46.1	32.8	-73.8
	12	-1	-53	397.9	227.4	-961.9	286.9	-229	364.3				
9	0	-7050.2	4.7	168.8	-40.4	51.1	-143.8	-141.4	-56.7	-159.6	79.5	-8.1	23.2
	12	2.3	-51.7	54.7	186.3	6.3	44.6	-91.9	54.7				
10	0	-329.7	-636.1	207.9	115	-178.2	-103.3	65	-120.6	-4683.1	-975.1	-77.1	85.8
	12	53.5	42	184.9	-51.3	355.2	-112.8	-321.5	79.8				
11	0	14.4	32.5	1394.3	-151.4	48.1	-35.6	293.8	4278.1	-118	173.6	21	-36.7
	12	-7.3	37.1	975.2	253.8	213.2	476.8	-1456	-23.3				
12	0	255.3	-5356.6	89	351	429.9	175.9	381.5	39.8	-315.5	43.2	57.3	196
	12	9.8	33.2	1035	92.4	1186	31.7	145.9	-27.5				
13	0	-2680.3	260.6	3250.2	-8.1	-326.9	12	-191.2	237.9	-30.4	-41.6	28.7	10.7
	12	-20.2	11.9	-456.6	146.7	50	-546	-1884.2	295.5				
14	0	144.6	-204.4	166.4	6474.3	-373.7	49	117	-49	-2.2	9.8	-7.6	-70.1
	12	88.3	-27.5	201.1	102.2	58.5	34	95.8	36.3				
15	0	953.3	23.7	296.6	91.9	50.3	-338.5	6794.1	7	0.2	77.2	27.9	96.8
	12	99.2	11.7	-48.9	7.5	138.1	39.4	426.9	26				
16	0	-1.2	-408.9	-306.6	-234.8	89	-555.9	22	106.6	98	11.5	-4.6	-15.3
	12	-52.1	58.4	11645.8	-104.3	-52	-66.8	162.1	-38.7				
17	0	-2503	-167.9	-131.7	159.9	75.5	-314.6	3624.7	-25.9	229.1	-144.9	30.6	-191.3
	12	51.7	-35.8	-69.2	-178.6	-196.6	202.2	165.9	-165.6				
18	0	14.1	-8789.4	-46	49.4	-14.4	112.4	-30.6	82.4	-160.4	21.3	74	50.2
	12	82.7	53.1	59.1	-15.9	-285.9	91.9	145.8	240.5				
19	0	-331.2	-1859.9	19.9	343.1	-63.4	3389.2	541.5	485.4	129.3	214.5	29.4	19
	12	33.4	-15.1	-1094.3	-528.6	266.3	-185.5	-478.7	-185.8				
20	0	-123	3966.6	-17.5	4279.8	-15.7	-47.5	239.8	76.4	9.8	-3.7	-2	29.3
	12	49.4	-109.1	182.2	-47.5	158	-80.9	542	-328.9				
21	0	-213.5	492.4	-408.5	2968.7	99.5	-651.8	-379.6	728.1	-230.4	192.2	84.5	-130
	12	-20.3	-78.7	-502.3	-74	-11514.1	-172.9	-882	-26.7				
22	0	-401.2	120.3	23.1	-233.7	3403	-332.7	47.4	23.8	190.7	-735.1	131.1	-83
	12	-30.7	-147.3	-263.5	306.2	-54.8	11629.9	308.2	-528				
23	0	-308.5	1931.8	659.7	126	-4247.4	-208	347.9	207.5	76	-238.6	184	-58
	12	-52.3	-23.7	194.3	165.8	46.6	400.7	-1299.6	-120.9				
24	0	-3467.8	-4196.2	-49	-28.7	-107.2	-172.7	-355.4	210.5	-72	-118.8	41.6	-159.4
	12	77.6	-97.3	-136.3	47.9	-402.7	132.7	-101.5	-288.8				

25	0	-71	517.2	-413.7	-1414	-374.7	-86.5	-285.7	322.6	-473	351.9	436.9	-71.8
	12	62.9	-4541.1	-314.8	271.7	25.2	250.9	-436.7	500.3				
26	0	94.6	-66.5	311.2	3960.3	3911.6	528.5	309.7	-213.5	74.9	52.6	28.3	-52.8
	12	-48.6	-101.2	-18.1	-283	671.4	-38.7	282.7	39.6				
27	0	692.1	-48.2	-87.8	112.4	43.7	7489	-30.8	-63.6	87.2	-13.4	59.8	25.1
	12	-82.8	4.7	-162	26.4	-94.8	-86	190.6	-353.1				
28	0	-4033.6	182.3	-237.9	-49	-181.6	453.2	-508.9	-56.1	-86.6	-51.4	90.5	-4.4
	12	-19.8	53.4	-97.7	-284.1	180.3	263.8	3747	0.8				
29	0	1019.4	-1791.3	15.8	252.2	83	-3434.4	132.8	-87.3	93.9	199.1	-149.5	-30.6
	12	14.1	31.4	-17.3	828.3	76.8	-403	21.1	541.3				
30	0	7.4	-0.4	10.4	165.8	-24.5	49.6	54	-88.5	96.7	-12001.5	-61.2	2.8
	12	43.7	-39.3	292.9	-87.4	-132.9	-36.3	538.9	45.2				
31	0	83.6	-75.6	37.8	-16.5	10805.3	65.4	-23.6	17	17.1	-73.1	21.4	10.9
	12	64	19.7	46	84.1	-112.2	-55.3	244.2	134.2				

Table 4.A.20 – Interleave VQ codebook 1 for core coder (WINDOW TYPE : LONG)

filename	Contents	mode	number of elements	number of vectors
cdmdct_1	MDCT	core LONG	20	32

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	235.3	67.3	50.9	-70.8	141	-11826.5	-180.2	-63.5	101.1	-67.7	-52.8	-26.3
	12	3.1	9.5	-213.3	91.1	-19.6	-208.5	185.1	155.8				
1	0	-2551.4	-264.8	68.9	-3066.5	-340.7	139.4	130.4	26.9	60.2	35.7	-4.1	74.7
	12	-35.7	10.4	-215.4	-28.2	182.3	258.3	272.9	-157.2				
2	0	625.1	-205.8	260.9	-388.8	-46.3	-37.3	40.9	-162.9	-59.8	-66.2	-57.3	-204.2
	12	-46.2	54.2	121.4	-154.5	-12125	83.4	421.2	168.3				
3	0	-209.8	227	-229.9	-47	6981.7	-61.2	43.9	-60.2	-68.3	50.4	34.3	-0.2
	12	16.8	10.1	-144.3	-9.7	-127.6	-128.7	-320.2	195.7				
4	0	3358.8	295.2	-171.2	-205.5	-2008.6	-196.7	25.4	-112.9	-50.6	-304.3	-65	-49.7
	12	-54	-35.7	-2190	54.1	-279	491.8	37.6	-327.2				
5	0	-11.3	-341.2	158.5	133.3	-158.6	2.8	-11.8	-38.9	6211.1	-519.8	-58.8	-26.8
	12	17.3	-57.9	341.8	-294.8	20.2	38.5	378.2	101.3				
6	0	66	85.5	1894.5	3096.2	-1994.1	40	-103.1	-369.4	-93.3	-65.3	151.1	34
	12	12.1	-27.7	-488.9	-136.9	39.4	1058.2	385.7	370.2				
7	0	-8	-8.6	-44.9	18.7	-73.3	50.5	-4	-61.6	-14.4	1.7	5132.2	-25.8
	12	147.1	42.4	18.2	2672.9	182.8	-251.4	-329.7	378.2				
8	0	-4.6	165.2	-55.1	9383.7	117.1	-95.3	-115.3	18.8	-0.1	59.2	70.4	-45.5
	12	21.6	-8.3	29.7	-51.5	294.9	-117.1	-209.5	-44.2				
9	0	-214.8	-80.8	47.6	-433.9	371.7	5001.4	-330.3	27.5	37.4	-44	-255.9	-153.2
	12	63.1	-12.2	-152.5	552.9	216.9	-145	-903.1	512.9				
10	0	1342.8	-240.9	-920.2	98.8	-132.1	-192.2	-158.8	3771.8	-21.2	-158.4	-48.4	107.4
	12	60.7	-10.7	-13.8	-227.2	240.3	478.1	319.2	496.1				
11	0	4.8	-5600.1	492	159.7	-55.6	-26.9	-318.6	121	149.8	91.1	126.5	-130.8
	12	-63.2	89.2	-228.1	-49	132.7	-101.9	95.8	-197.4				
12	0	356.8	2654.8	-358.9	114	103.3	85.6	22.1	-37.8	106.3	1.2	-15	-49.8
	12	-72.9	170.8	-66	36.2	2617.9	-162.5	503.6	1981.6				
13	0	-9.4	127.7	23.1	-66.9	-122.5	-198.8	101.7	56.8	41.2	89.5	-60.9	30.3
	12	5664.6	159.7	213.8	-162.9	-59.2	84.3	-33.3	-259.1				
14	0	-103.3	42.4	-1	-180.8	44.4	743.2	4688.1	28.2	-81.9	51.3	69.5	61.2
	12	-106.5	-11.6	-37	-126.5	-27.4	164.2	-136.7	-88.4				
15	0	-1042.2	-313.7	-154.2	-414.9	650.3	-2755.9	-161.8	364.9	103.8	27.5	-94.9	-172.6
	12	-75.1	108.1	-7700	-310.2	161.1	-56.1	-341.6	-34.1				
16	0	-54.2	-3008.6	-3909.4	-85.6	-62.9	-56.7	108.9	40	267.7	-151.7	144.4	-63.7
	12	-221.4	-67.8	193.4	55.7	572.8	-265.8	-565.6	-407.7				
17	0	-63.3	231.6	-3415.5	601	575.7	1192.1	-742.8	47	-427.7	29.6	58.8	46.8
	12	-87.6	101.1	-843.8	314.8	-1669.2	-178.3	-315.2	-230.1				

18	0	558	-233.4	130.9	5.3	42.9	-227.1	63.8	-7070.4	-35.4	27.3	-70.9	-14.2
	12	30.6	127.3	499.7	11.2	343.1	789.6	860.6	325.5				
19	0	-3560.6	3198.6	-27.2	19	3.2	-363.7	52.9	257.5	-150.4	128.9	-159.8	-58.2
	12	1.2	17.3	-15.3	28.7	-376.5	-186.5	121.6	-72.7				
20	0	-1971.3	-21.1	-60.6	3407	-171.8	35.7	-15.5	65.3	173.8	-58.9	2.5	70.6
	12	-48.5	-24.8	306.4	156.2	476.7	-327.6	-2828.6	-905.5				
21	0	-580.4	-214.9	-1765.9	76.6	-4029.4	-5.1	-172.3	-113	51.4	-152	15	-37.4
	12	27.1	-56.2	266.6	-408	589.2	298.9	941.4	-136				
22	0	297.5	-16.5	62.7	-49.7	3.1	97.1	-11428.2	64.4	11.5	64	32.8	-114.3
	12	2.9	10	-32.2	117.6	-48.1	68.9	293.1	14.3				
23	0	-49.9	2192.6	578.3	-4411.8	-44.1	134.5	-64.7	55.5	-10.8	-3.7	-90.7	3.1
	12	-46.5	92.6	431	139.3	-407.1	15	-167.1	-108.2				
24	0	28.8	-7.1	121	-63.5	79.6	-35.6	-14	45.6	-201.3	5649.4	-13.3	-12.9
	12	43.6	26.2	155.9	-306.9	-134.7	22.8	556	-21				
25	0	6.7	-62.4	-23.1	790.1	199.2	-93.1	72.3	-203.9	-26.1	-374.6	-437.6	219.8
	12	-139.7	-9142.4	30.3	-176.5	119.9	-146.4	-595	-391.8				
26	0	2816.7	122.3	3753.5	-33.5	242.4	245.3	233.1	-340	17.7	-215.5	40.5	-58.3
	12	6.6	29.7	-254.2	-2.6	675.4	-347.6	506.2	227.3				
27	0	-682.7	-375.4	5715.8	196.7	236.8	214	21.2	229.6	-232.8	-41.2	-54.1	2.8
	12	-74.8	-148.1	-103.5	-55.2	298.6	-263.6	393.1	-462.4				
28	0	-10807.7	112.1	-283.5	173.9	368.5	172.3	46.9	208	51.4	226.4	72.3	108.1
	12	-76.5	37	543.5	-277.6	-291	-178.8	-246	228.4				
29	0	-58.2	-42.1	-36.1	-25.6	-78.7	315.2	193.4	11.3	47.5	46	44.7	12181
	12	-7.5	-10	-31.3	106	255.7	78.9	571.9	-518.5				
30	0	-102.9	20.3	-9260.7	53.2	55.9	133.6	125.5	202.7	-66.6	-62.6	-7.3	2.2
	12	-0.6	56.8	-6.7	-14.7	-340.4	-160.4	-419.5	-241.2				
31	0	5816.2	60.8	-128.6	-76.2	350	99.9	-81.6	133.1	-103.3	-19.4	56	-81
	12	-87.3	-14.7	276.2	194.7	-148.2	129.3	-495.4	-270.1				

Table 4.A.21 – Inteleave VQ codebook 0 for core coder (WINDOW TYPE : SHORT)

filename	contents	mode	number of elements	number of vectors
cdmdct_2	MDCT	core SHORT	17	32

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	16384	-534	296.2	242.6	-1011.7	-156.3	-25.8	-720	135.3	-288.9	-197.1	339.2
	12	12.6	484.7	68.3	1615.2	-524.1							
1	0	-7.6	-264.3	-165	453.1	-8874	-37.8	918.3	-257.6	183.8	-144.9	115.8	63.6
	12	352.4	137.9	322.3	-630.2	-5343							
2	0	15.7	337.7	-1005.3	4231.3	-2785.4	214	-3.1	311.3	-296.8	99.8	-496.3	186.7
	12	-33.9	-685.2	642.6	633.4	4888.4							
3	0	149.2	-252.5	-238.4	-191.4	45.6	154.2	-498.2	-4264.6	-1516.9	985.7	-939.2	-419.5
	12	1279.1	581.1	-1809	-951.5	1015.2							
4	0	239	-72	-104.5	882.2	-73.4	24.2	-187.9	141.5	-9608	337.9	253.2	433.7
	12	-533.1	-1421.2	-2313.7	877.7	-1429.2							
5	0	233.8	-356.3	-704.4	218.4	-766.8	-226.7	-7255.4	-139.7	229.3	-439.2	-523	230.3
	12	-173.3	1689	-806.6	2720	-1824.9							
6	0	-651.9	677.9	-425	-7670.7	-637.7	168.6	-751.9	-37.3	-471.9	193	137.8	293
	12	367.1	-910.5	-542	-4284.2	5243.1							
7	0	-45.2	609	-135.9	-740.3	2550.1	3678.4	1197	774.5	-1783.1	355.6	-191.8	-292.4
	12	-293.4	-196.9	199.3	-2391.9	429.2							
8	0	819.6	-838.5	-374.1	-704.7	-329.2	310.2	404	-204.9	94.1	1035.5	-487.3	1144.9
	12	-0.1	-8832	1367.5	120.2	-1549.2							
9	0	-274.8	-263.5	590.7	723.8	420.4	-272.4	346.6	-478.9	3294.9	1890.3	-1243.6	-200.1
	12	96.6	232.9	-3382.5	953.2	-1167.4							
10	0	-113.3	-594.2	-9768.3	80.5	-315.6	12.5	281.2	-97.8	-37.7	427.1	310.7	-30.2
	12	-90.6	92.8	909.2	-684.6	-954.4							

11	0	-92.3	141	576.4	-24.4	56.3	-1025.6	115.2	-8990.2	176.6	144.3	86.6	-190.4
	12	-1808.1	-673	452.2	-29	3351.3							
12	0	152.2	2705.2	2093.3	-1395.5	27.3	-76.7	835.9	-54.8	-363.6	690.4	-316.8	-151.5
	12	-539.3	492.3	2115	777.4	-3297.1							
13	0	-266.9	-272.8	-464.7	262.5	0.4	11.9	441.1	-267.7	-33.5	361.3	4513.6	-476.3
	12	12.5	-138.8	-965.3	2941	-760.6							
14	0	-5805.6	289.6	469.5	1712.9	-169.3	150.8	-1835.6	734.9	90	551.8	-236.2	-323.6
	12	-172.4	-5589.5	-2574.5	-1050.4	1301.2							
15	0	-3543.8	329.1	-598.6	-274.4	1067	973.2	856.7	549.2	-31.2	-82.9	-907.1	671.3
	12	-115.7	2434.7	2131.6	-931.5	-1369.2							
16	0	-355.6	242.4	-267.9	182.5	177.8	-134	85.2	806.1	117.7	-121.4	-738.8	-6574
	12	-813.8	-615.8	-632.4	390.3	1416.5							
17	0	-18.8	247.7	2109.5	-2071.1	-3397	1029.1	474.4	-52.4	620.7	63.1	518.2	-206.5
	12	288.5	1795.3	-2903	-2856.7	1327.6							
18	0	260.8	496.6	695.4	-1262.8	409	-521.2	-2073	1709	1929.4	69.5	-1172.1	-1298.6
	12	3453.2	-77.8	-1448.3	1117.8	1309.2							
19	0	3500.7	-3781.1	337.1	350.8	620.7	-166.5	507.2	-305	-15.5	-51	-343.1	310
	12	21.2	1445.3	1738.2	-576	1721.7							
20	0	-32.2	-54.2	-1286.9	76.5	-1713.8	-455.4	3055.5	1436.6	345.3	-55	94.7	-580.7
	12	-39.9	1640.6	2128	625.4	-2599.5							
21	0	-178.5	-715.7	-315.2	-671.3	-1472.1	4815.1	-753.8	83.8	317.6	136	559.3	-192.3
	12	690	-2273.6	-2537.7	1253	-3324.6							
22	0	591	-267.2	-299.1	-485.2	-423.5	-38.2	674.1	-2357.8	1102.6	-3306.7	-757.9	-302.3
	12	564.8	2143.9	-754.7	2681	611.6							
23	0	-525.9	-124.6	-189.3	111.3	427.3	-505.8	-420.1	127.8	-364.7	-7157.5	-47.4	226.7
	12	-380.5	195.8	-1269	-18.3	-204.7							
24	0	-492.2	-4261.3	1940.7	1797.6	341.4	601.3	611.3	27.5	252.7	151.2	390.8	-54.1
	12	-1216.6	2894.2	-3176	-1731.3	-1433.6							
25	0	-12834.3	-93.1	-374.5	287	-1899.4	35.7	-715.3	-307.9	10.6	78.9	1096.8	34.3
	12	103.6	2076.4	615.2	1465.3	758							
26	0	1855.5	683.3	-272.6	-279.2	818.5	-796.5	85.7	57.3	161.8	-7.5	271.7	-197.5
	12	-6743	2581.4	-1422.1	175	-1154.1							
27	0	-129.1	-379.4	-4225.6	-2308.9	247.6	246.7	-478.2	234.3	327.2	-193.4	-380.8	165.5
	12	-322.4	-1003.3	-706.7	135.4	2359.3							
28	0	-8970.6	625.4	316.8	-108.1	625.2	1014.1	934.6	-815.1	-137.6	-365.9	-392.8	-204.2
	12	-130.7	-195.9	1439.7	3056.3	603.4							
29	0	-90	1303.7	-1332.6	167.8	-600.4	2072	-1718.3	213.6	125	-37.1	21.4	106.7
	12	-3498.4	2942.1	-421.4	1483	5828.9							
30	0	478.7	7218.4	129.5	141.4	732.8	36.2	296.8	-233.7	72.8	121.4	-214.2	-431.6
	12	2.4	741.3	1486.1	1177.2	1714.4							
31	0	-4432.8	-852.9	-2584.9	-89.8	-780.6	-722	-88.5	-1466.4	32.9	161.5	-65.6	-642.8
	12	207.1	735.6	-5723.4	-841.8	-361.4							

Table 4.A.22 – Inter leave VQ codebook 1 for core coder (WINDOW TYPE : SHORT)

filename	contents	mode	number of elements	number of vectors
cdmdct_3	MDCT	core SHORT	17	32

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	210.8	-13952.3	-184.9	-354	562.2	506.7	-239	-758.3	-199.4	205.1	32.8	-1105.2
	12	-674.6	577.5	-224.6	-325.9	-1149.9							
1	0	-413.1	83.5	-336.7	-1688	-1187.9	-3898.7	-171.5	897.3	-485.8	695.3	643.6	-319.2
	12	221.8	-3611.9	3212	-1658.3	-2700.4							
2	0	-331.7	418.3	-46.9	126	865.1	463.8	-6325.3	271.8	176.6	-314.8	-18.7	-555.5
	12	795.3	-474.9	628.7	-2206.3	55.7							
3	0	-4269	-392.4	3219.7	-539.8	14.4	-774.7	-262.6	-302.9	46.9	70.7	-63.6	120.8
	12	59.4	811.7	4138.9	-9.2	112.3							

4	0	767.6	-710.4	-7343.7	108.3	660.1	-256.2	-137.4	-77.8	-77.4	276.4	-25	325.9
	12	46.3	-578.7	2070.7	1799.3	1214.2							
5	0	-141.8	3251.9	-1072.7	343.9	-593.7	-29	191.1	1221.3	-369.9	-22.8	-530.7	1110.2
	12	-1079.3	-47.4	-3613.6	-3544.2	1237.2							
6	0	-241	877.9	-36.2	125.9	253.3	8498.6	-447.2	-525.9	685.6	502.4	-196.4	1546.3
	12	755.3	382	305.2	-797.4	-1413							
7	0	849.4	-216.3	375.9	-12034.9	-438	-444.1	250.6	80.4	1147.9	193.8	413.3	66.6
	12	-56.1	1401	-1495.1	-445.3	-475.5							
8	0	3884.2	4137.2	298.4	3.6	326	225.8	367.8	-384.3	147	-7.4	358	-281.7
	12	447.3	423.2	1928.4	-3456.7	-2995.4							
9	0	482	292.9	-1031.8	47.7	-3246.4	248.2	666.6	-548.3	-470.8	430.4	-840.3	-332
	12	156.3	-5991.7	-387.9	-641.7	-316.6							
10	0	-231.9	-198.8	-307.8	452.8	12.6	-418.5	88.7	-41	358.8	-388.4	-428.7	4450.3
	12	384.5	737.2	-4.5	4933.9	1082.1							
11	0	5421.3	346.1	127	3133.9	-586.4	76.3	585.8	-319	147	-67	-700.3	206.2
	12	93.4	-4386.2	443.2	682.7	2974							
12	0	-303	-58.5	-133.6	609.8	-339.1	-79.9	112.4	577.4	3997.3	-635.8	275	-193.2
	12	573.9	1187.4	1662.3	1321.8	527.8							
13	0	102.2	-1280.8	-539.2	-4552.8	-744.6	694.4	-49	94.7	-22	-5.5	-577.7	-254.2
	12	114.7	-531.6	-3841.1	900.7	3404.9							
14	0	-573.8	117.5	-2	259.8	-324.1	-48.6	-1858.6	-3521.6	94.6	-161.4	908.1	-404
	12	-416.5	361.2	-792	-303.6	-1569							
15	0	128.5	-2026.6	78.6	-476.9	-975.5	-358.2	-1236.6	3351.7	-75	44.7	636.2	67.4
	12	6.3	2449.2	-374.2	-126.5	-3959.9							
16	0	-30.5	-640.4	1339.6	1507.5	-493.6	-403.1	22.8	-368.6	-1413.1	-2039.5	-1884.3	-1548.5
	12	-654.6	2228.7	545.6	1654.6	-4512.5							
17	0	-9067.5	-617.7	-460.9	264.7	1549.6	54.8	-82.9	204.5	22	90.4	120.8	421.4
	12	-36.4	-445.4	524.9	811.8	1014.5							
18	0	6359	-1291.6	-175.5	-279.3	1966	183.8	-1376.2	-322.4	91.9	124.1	987.5	-377.8
	12	466	2217.7	-1671.1	-1155.7	-88.9							
19	0	-108.4	-170.5	-2132.2	3481.2	-207.9	1148.2	-640.2	202.6	191	324	103.3	-949.2
	12	3242.5	-123.1	129.5	-2247.7	125.2							
20	0	18.2	-309.1	-693.2	-63.8	5058.6	1085.2	1284.9	-674.8	395.6	253.9	-412.8	-290.1
	12	37.7	-810	1208.4	-624.7	-8533.2							
21	0	66.8	103.7	136	296.5	275.6	86.2	-373.8	-81.1	5	-404.3	6575.4	-38.4
	12	417.6	183.3	1674.7	-342.1	1390.6							
22	0	97.2	-6.6	764.7	417.9	339.4	-160.5	-22	-560.2	16.2	4354.3	-714.5	-186.2
	12	-792.8	5761.3	-2868.7	426	264.2							
23	0	-267.9	-69.1	165.6	-718.6	-826.1	2625.5	3479.2	511.1	528.3	172	259.7	-380.7
	12	507.4	278.7	803.8	373.8	870.7							
24	0	598	-1202.7	5005.8	-1061.1	1246.4	27.8	-6.3	358	-147.8	-604.8	-50.5	-175.6
	12	-131.6	-2995.7	-1675.1	-1108.7	1871.5							
25	0	-2505.2	-282.2	145.1	399.2	1227.1	-1103.3	1568.9	-203	-65.6	104	427.1	-180.8
	12	102.3	-109.7	-2583.3	1662.1	1983.7							
26	0	60	3970.3	4094.5	844.5	-1026.5	114.6	-830.2	-38.5	568.2	-555.8	-49.7	242.7
	12	853.8	-1240.9	151.6	418.9	485.3							
27	0	58.5	56.9	156	37.8	-305.9	-437.6	548.6	651.9	-6621.1	284.5	-295.6	-333.5
	12	1616.7	1988	1497.3	-51.3	2570.1							
28	0	-275.8	8445.6	-1183.1	849.5	-337.1	48.2	-161.5	-270.5	431.1	24.4	507.4	17.2
	12	-571.8	580.6	-2151.6	674.3	-1505							
29	0	-43.4	-348.7	2831.3	459.9	-784.6	1878.2	-188.4	-91.6	-819.7	1119.6	344.8	709.9
	12	160.9	1134.7	-777.8	-431.3	63							
30	0	120.6	-343.5	15.8	-248.1	143.5	528.9	761.2	-6620.7	476.2	-192.8	669.9	546.6
	12	467.6	-65.3	2492.2	916.1	-781.6							
31	0	1232.8	-125.6	-52.2	-361.1	669.9	-600.5	359.3	125	-195.9	-437.1	-282.8	216.6
	12	6506.1	1893.6	-117.7	-55.5	7375.8							

Table 4.A.23 – Interleave VQ codebook 0 for scalable coder (WINDOW TYPE : LONG)

Filename	contents	mode	number of elements	number of vectors
scmdct_0	MDCT	enhance LONG	28	32

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	128.3	-932.2	-269.6	-417.6	-94.2	340.3	-83.8	-8130.1	144.4	249	156.5	-315.9
	12	-95.4	404.7	-662.3	-283.8	94	208.8	-97.2	-56.2	97.9	396.1	182	-135
	24	-38.2	306.2	91.3	106.5								
1	0	201.2	-4276	1196.9	392.1	-5216.8	598.6	376.6	131.8	56.3	-2237	894.1	-832.2
	12	-256.5	-506.3	481.2	321.3	393.9	-91.1	103.8	452	170.5	235.3	269	459
	24	166.1	-664	-17.3	339								
2	0	2617	85.4	-167.7	-1804.1	4414.1	498.8	1432.6	-1359.2	632.5	-4937.2	-415.2	174.3
	12	-1008.8	300	112.1	70	158	-152.1	275.7	-391.9	50.5	136	-80	-413.4
	24	9	468.8	46.9	325.5								
3	0	301.2	137.9	-1784.8	45.9	-823.1	4.9	144.8	154	322.9	156.4	51.7	-162.2
	12	275.2	319.9	-146.4	-2.5	221.7	35.6	-16.2	688	148.6	348.9	8192	-112.7
	24	-629.1	218.2	373.9	72.8								
4	0	-3885.1	-446.4	-519.5	566.8	662.8	-1059.8	5754.8	251.3	-416.1	-798.9	-728.2	697.9
	12	-679.3	-473.4	-192.3	-159.6	-119.9	-81.8	-66.8	-37.8	-143.7	-33.9	-20	40
	24	-93.6	-47.5	-192.2	-117.1								
5	0	754	275.7	-345.7	129	259.6	-803.1	-85.8	374	-198.4	323.5	470	1314.9
	12	956.8	-319.2	-30.2	1585.1	-12.5	-123.9	-8192	-343.6	-109.5	-101.9	130	-686.1
	24	384	-180.1	-413.2	-23.9								
6	0	2396.1	355.6	-207.3	1833.8	-10.6	-1321.6	7373	-158	144.5	-284.1	93.2	-317
	12	-545.6	-237.8	-271.6	-132.6	91.9	-17.1	-94	374	460.8	-193.5	-409.3	92.1
	24	-235.6	88.2	96.8	-121.1								
7	0	-651.8	-2127.6	655.4	7212.4	-323.1	-477.9	1955.8	303.2	-420.9	1250.9	-27.1	-552.6
	12	59.2	-45.7	71.2	-178.6	107.9	-89.2	88.7	-236.6	-200.7	253.5	-280.3	265.6
	24	-188	-223.3	-172.2	338.6								
8	0	-353.3	-490.6	53.3	1929.3	-386	7211.5	669.8	-267	-1117.5	-390.9	-264.1	583.9
	12	1183.9	1311.3	-79.4	-333.2	24.9	159.3	-210.4	116.9	-194.3	52	-82	-105.1
	24	239.1	134.9	-69	-116.3								
9	0	14.3	348.3	868.3	-63.9	321.9	-2221.2	53.3	215.6	-7890.4	-3.5	233.1	163.6
	12	499.4	131	-0.4	-131.6	13.7	-230.4	221.1	-61.7	-189.2	-252	117.3	426.7
	24	89.1	-323.3	209.5	16.4								
10	0	-221	-173.4	59.4	-119.5	116.1	-19.6	45.3	96.6	152	-19	195.5	141.4
	12	2.7	-32.6	125.8	-104.3	-8192	12.6	-132.6	264.3	-136.2	-180.6	76.3	-98.8
	24	141.3	-164.8	-325.5	128.1								
11	0	-3222.5	377.5	4063.5	834.4	1458.5	2100.7	378.5	73.9	-311.6	-200.6	-139.7	-125.7
	12	581.7	259.8	151.5	82	-30.6	-35.4	-247.7	402	6035.4	-166.7	-100.8	-75.2
	24	284.6	-53.3	-318.5	232.8								
12	0	1721.7	-4502.2	-852.4	1847.9	46.2	-228.1	1950.1	-221	-671.4	1833.4	-4362.2	1558.5
	12	-1244.3	20.6	34.9	294.9	144	202	-90.3	-813.7	-142.5	117.9	260.1	-37.7
	24	1205.4	-476	-1403.6	36.6								
13	0	-7301.4	1080.6	-901.3	97.1	-380.6	996.1	-699.1	1045.8	-514.7	920.1	283	-1027.7
	12	-166.3	371.3	190.4	89.6	-808.9	-141.1	118.3	169.7	-481.7	-209.4	92.2	27.4
	24	374.3	-135.2	477	15.8								
14	0	-147.3	4031.9	1435.8	-46.5	-5117.2	81	885.1	-1175.5	-1107.2	-887.5	-763.1	504.5
	12	-304.1	419.4	-131.7	-258.4	44.3	-169.2	-284.9	-656.5	-168.8	249.9	92.2	434.8
	24	399.3	2742.8	-2269.8	-422.2								
15	0	55	2056.7	41.7	644.4	-94.3	-222.1	403.9	173.3	78.1	285.3	213	-356.7
	12	-415.1	-64.4	8192	27.7	129.3	-171.8	124.2	-475.3	-119.8	553.9	168.1	405.4
	24	231.8	-115.4	357.8	-11.8								
16	0	-4373.6	-4707.6	-234.2	-154.3	-32.1	544.7	-566.6	-793.3	514.2	542.4	-440.2	-49.7
	12	595.3	-94	102.5	178.9	-85.8	-285.5	-218.5	143.9	31.4	98.4	180.4	-214.9
	24	3.5	262.8	-484.5	-88								

17	0	290	-8192	-278.4	297.9	146.5	-0.5	-588.2	-248.6	-199.9	-520.7	497.6	475.9
	12	118.3	-89	107.1	-91.6	-86.8	91.8	-77.5	233.9	-58.3	394.4	-145.9	-215.6
	24	265.4	-149.4	-72.8	135.5								
18	0	-110.7	-1113	-290.1	4806.1	1204.9	-730.4	-5190	400.1	-593	-1403.1	-340.4	677.9
	12	29.2	-122.1	20.3	320.7	79.6	181.3	373.8	20	-197.5	-354.6	-208.2	129.6
	24	-98.4	103.4	252.8	-91.8								
19	0	378.7	137	-1140	-592.3	-251.4	-532.3	1373.9	238.4	1098.1	-27.9	403.4	32.6
	12	8140.6	-375.4	-137.7	-83.2	-147.7	168.8	57	360.3	179.8	-195.7	-220.1	-226.8
	24	170.5	230.1	-163	-56.9								
20	0	-987.2	30.1	866.7	526.6	8093.1	70.7	-55.7	-699.4	-1164.6	-316.1	1375.7	189
	12	22.1	-498.7	347.6	-16.3	22.9	-76.5	0.5	97.8	-269.5	74	-329.3	416.5
	24	51.1	-208.5	-687.8	-355.3								
21	0	630.7	-1598.8	-450.4	-489.3	168	1335.9	-243.7	841	-433.4	-23.2	561.6	413.4
	12	17.6	-575.4	49.2	-282.7	-210	-28.5	12.3	228.6	70.7	7983.5	43.6	-92.4
	24	-191.5	124.7	-265.7	-638.3								
22	0	-890.3	548.5	-274.7	1330.1	-126	104.6	497.7	85.7	217.6	21.2	8192	370.7
	12	-722.8	0.7	-477.6	20.6	1.1	-385.2	46.2	-386.8	-204.8	-104.7	-154.2	-134.7
	24	131.3	-264.4	-509.6	-312.1								
23	0	-3520.7	1720.9	-920.1	-958.3	633.2	-725.3	-1123.3	-240.1	2405	-1595.2	-2771.2	-464
	12	-995.5	-2664.6	-962	-1557.6	529.4	176.4	-432.5	26.9	-311.8	68.5	-222.6	162.7
	24	4052	-508.5	-260.2	-5318.8								
24	0	-1135.5	-686.1	-738.4	27	225.8	-2268.9	219.7	608.7	276.7	586.2	-316.9	43.1
	12	-28.4	8192	415.8	-84.4	174.8	-61.7	-40.6	-286.4	148.3	101.6	-161	-82.4
	24	110.8	268.8	-79.3	121.3								
25	0	-311.8	-137.4	-670.6	-884.3	113.8	-261.8	-146.9	-241.6	-489.4	155.7	-64	8192
	12	-116.1	-97	-146.1	357.8	7.3	254.2	220.7	-1634.1	353.3	-134.5	-26.2	-222.5
	24	-382.3	4.3	112.5	-157.4								
26	0	1862.8	-495.5	6418.9	-149.9	340.6	-2238.5	196.6	-164.2	-1437.8	435.9	710.5	-514.6
	12	169.9	-348.1	-74	106	138.6	-177.7	21.5	283.2	-545.6	89.3	-63.1	431
	24	764.4	-249.3	-414.9	-587.9								
27	0	-41.7	270.5	-359.4	-10.9	-27.3	-261	142.7	77.9	14.5	19.9	345.9	52
	12	-78.3	81	122.3	102.6	3.7	8192	-30.7	-356.9	-97.6	73.5	-216.4	-302.7
	24	-153.8	42.4	2.2	98								
28	0	-583.8	365	-4062.4	1.6	1181	833.6	650.5	-1345.9	-5367.4	231.7	93.2	-2805.4
	12	366.5	-317.6	-158.5	116.1	-199.2	-92.6	223.6	817	174.4	-114.9	17.9	-1200.2
	24	148.8	-180.4	-170	-80.1								
29	0	-1043	-79.6	-359.2	-298.3	-661.8	-1090.7	128.3	-837.1	-1261.5	-8192	216.3	141.4
	12	-51.2	582.6	205	182.4	-69.7	335.1	110.7	118.4	-108	-75.7	274.5	-77.5
	24	18.7	-89.6	179.5	-99.4								
30	0	709.8	-1626.7	-165.6	1293.3	19.2	-1666.5	133.5	138.4	-806.8	-440.8	518.5	696.7
	12	75	129.9	-23.3	-8192	38.3	-374.9	-127.1	-946.8	129.2	-201	-42.1	251.4
	24	293	430.6	-511.5	637								
31	0	-710.9	4344.6	-134.9	5527.1	-120.1	713.5	394.8	-680.5	404	66.3	336.9	478
	12	-302.1	-30.3	27.7	318	-498.2	23	85.9	-151.9	-74.7	-77.4	-225.1	242.8
	24	780.9	-393	656.9	715.2								

Table 4.A.24 – Interleave VQ codebook 1 for scalable coder (WINDOW TYPE : LONG)

filename	contents	mode	number of elements	number of vectors
scmdct_1	MDCT	enhance LONG	28	32

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	-561.2	-713.8	-16.7	1735.2	215	-8119.3	-300	-116.6	-321.4	830.9	355.2	-58.3
	12	280.3	464.6	167.6	47.9	-11.4	96.1	137.8	-252.5	137.8	-303.4	-98.5	14.5
	24	-225.3	160.7	-12	-58.6								
1	0	-775	315.3	-620.2	-992.4	-12.8	-1335.7	143.6	-1198.4	-1288.1	8015.5	204.3	158.9
	12	-591.5	731.1	144.8	179.3	93.2	145.5	38.9	257.6	101.6	-67.5	50.9	11.4

	24	99.4	-210.4	-40	-58.5								
2	0	-3915	-782.6	-1029.4	4961	534.9	-181.7	606.6	369.6	-1139.2	-1963.3	-426.4	-941.3
	12	259	276.8	121.6	84.6	202	449.1	144.5	205.4	-366.1	-335.5	20.2	82.8
	24	-357.4	119.7	-107.2	-358.5								
3	0	-1610.2	-1283.8	-1449.3	437.8	-267.9	5562.3	-2744.8	-305.7	278	2618.1	326.5	-169.6
	12	-2019.1	-1045.5	321.3	281.7	-564.1	159.5	358.6	-793.5	-231.2	-47.2	172.5	195.9
	24	-473.5	609.3	-268.5	-18.1								
4	0	203.1	1157.9	-332	8192	246.5	-60.8	-900.1	285.3	878.8	-932.6	72.4	223.6
	12	206.3	117.7	195.7	-205.6	50.3	-217.2	-262.4	197.6	33.2	-89.9	313.1	270.3
	24	-78.6	208.8	135.8	-92								
5	0	-182.8	3165.2	500.3	-3345.9	631.5	-220.1	-219.9	4608.3	-3178.4	-399.9	-312.2	378.8
	12	-559.3	196.1	354.2	24.5	397.2	24.5	-122.3	-579.3	-29.6	-351.2	29	306
	24	-518.6	329.2	259	-361.8								
6	0	3560.4	-556.8	901.4	-1009.7	-3115.6	-1781.1	20.4	-448.8	-2473.8	-1018.8	-3821.8	-655
	12	-247.1	231.3	-350.5	-76.9	-2524.5	-11.8	601.6	-42.9	503.4	11.8	-98.8	563.1
	24	-855.1	-613.3	877.7	-836.3								
7	0	99.3	803.3	-350.6	-103.9	556.6	-2584.9	501.2	24.7	7101.9	42.6	83.8	-947.8
	12	630.4	491.2	-69	3.7	-182.6	-165.1	372.6	64.4	120.3	-103.6	51.5	3.1
	24	-483.5	-39.7	254.2	315.2								
8	0	-684.1	-186.1	1408	467.3	516.3	647.9	-1823.9	-209.9	-1533.2	214.2	-735.8	-223.6
	12	7325.9	349.4	210.1	66.3	160.8	63.6	-153.7	-434.8	-205.1	214	479.9	138.6
	24	-90.4	-36.2	260.9	-82								
9	0	326	510.6	664.5	933.7	-171.2	614.6	264.1	528.9	97.5	-301.1	157	6638.7
	12	-70.2	209.6	74.7	-734.7	91.6	-496.3	-258.8	4242	-162.9	-179.7	101.5	346.6
	24	250.4	-181.2	165	-184								
10	0	536.5	1332.9	164	1325	-231.2	-62	-8136.5	13.3	-8.8	-183.4	-21.5	-156.9
	12	-748.1	78.5	-84	-435.3	12.9	-236	-118.6	182.9	-26.3	52.5	-93.9	402.3
	24	-43.3	-265.1	249.3	96.7								
11	0	2532.6	-1240.8	4875.4	-395.9	744.2	4754.6	-212.9	538.6	38.7	-791.5	311.9	-400
	12	-141.9	-34.1	122.9	20.4	310.5	289	-264	34.3	-295.5	13.5	-98.2	85.9
	24	155.5	-421.3	-420.2	-1113.5								
12	0	-854.1	-4737.3	-4964.8	-42.6	36	271	402.8	216	-361.4	-1031.3	896.6	303.3
	12	-882.9	146.9	407.8	-57.6	140.3	-62.1	-208.2	-299	62.6	-104	-54	-22
	24	125	541	167.3	62.6								
13	0	-37.3	-320.7	-479.8	-1002	6982.4	428.5	429.6	1185.6	270.6	-38.5	-2430	-216.1
	12	-258.1	1022.9	7.5	31.9	145.5	-95.5	57.1	-248.5	-362.5	386.9	-647.1	-296.8
	24	-215.8	657.7	158	-170.8								
14	0	-936.6	-6447.3	1963.8	-43.3	621.1	-589.6	790.4	34.8	192	-681.5	-75.1	-1008.5
	12	-592.1	372.8	-407.7	-232.1	84.5	48	735.8	-190.7	-189	-36.1	-20.3	69
	24	-379.5	24.9	-290.1	-187.7								
15	0	-903	-538.2	-466.7	82.9	323.1	-2047.7	296	732.5	-316.9	625.5	-87.2	19.2
	12	117	-8038.9	371.8	126.6	-22.1	-100.3	-177.6	-126.4	260.5	0.5	0.6	34.8
	24	-285.4	103.3	-347.5	389.4								
16	0	1747.9	2234.3	-277.5	-109.8	-171.4	-878.5	-34.1	-6668.5	-277.8	-71.8	-929.3	77.1
	12	-55.3	-834.8	1929.4	4.5	18.3	59.5	-95.9	305.9	158.7	59.7	248.2	-149.5
	24	128	-65.1	-1.6	213.8								
17	0	-739.1	-212.2	400.8	70.1	-189.8	683.7	40.3	-532	3	-695	-1249.8	-1958.4
	12	-1494	142.9	-97.3	-2738.4	-112.9	159	-7679.3	260.8	-267.6	-203.5	147.6	1002.2
	24	-304.2	317.3	792.7	427.9								
18	0	3208.6	-221.6	-1632.8	444.5	-382.3	-171.5	130.6	5996.9	1416.9	299.8	-641.1	-1334.1
	12	-135	-295.7	999.2	-588.2	-109.4	0.8	45.1	241.5	716	75.3	181.6	-63.4
	24	608	302.1	-2.8	186.6								
19	0	-4161.7	-205.9	-505	-5407	132.5	913.8	441.8	-43.6	281.5	-884	369	306.5
	12	236.6	16.9	321.3	-136	-285.3	-57	117.7	172.7	-341.1	-286.4	518.2	-36.8
	24	-165.2	38.2	-128	2036.2								
20	0	4768.2	651.6	-1323.6	-21.5	3555.3	2592.6	672.9	-1588.3	776.8	1531.6	170.8	-1284.5
	12	-351.6	29.5	637.3	-571.3	401.2	-84.3	-197.4	-192.6	304.2	-91.6	-144.7	-112.9

	24	135.2	-150.4	49.5	492.3								
21	0	566.2	193.6	-8192	-1008.5	311	-703.7	143.8	-322.4	-345.2	493.5	-122.1	-583.3
	12	-72.1	189.9	-399.4	-105	209	-118.6	97.1	220.5	479.3	-14.2	198.2	-104.4
	24	202.3	83.1	12.5	226.3								
22	0	-529.2	317.2	-3662.3	-2057.5	-4863.5	1738	495.1	783.9	9.1	105.2	-514.3	84.7
	12	258.6	538.8	134	-14.6	359.9	-259.7	-39.4	-164.9	525.2	19.7	-3801.9	78.5
	24	-237.3	-526.4	466.5	20.8								
23	0	1203.4	-1301.8	348.1	-1694.5	-83.5	-667.8	-69.4	-248.8	-806.8	-32.7	7341	-343.4
	12	80	-278.2	174.6	-342.1	-312.2	498.4	-382.4	-16.3	252.4	246.8	1.9	111.2
	24	321.4	77.7	248.6	-645.2								
24	0	-2744	-226.8	5684.9	-3196.3	201	-1143.2	-68.9	48	625.6	-12.2	-1007.2	-911.8
	12	-619.4	413.5	-113.2	45.1	334.9	15.2	-7.5	363.2	-1040	-165.1	230.6	-309.3
	24	510.4	-149.2	-171.9	454.3								
25	0	482.8	2739.8	-178.6	648.8	19.1	-422	129.6	155.7	-32.2	218.2	255.1	-766.4
	12	-511.5	-441.8	-7895.9	-95.4	88.4	-507.1	-117.5	-160.8	110.5	478.6	-43.8	375.2
	24	-422.4	-239.4	273	43.8								
26	0	459.8	2520.3	-3700.9	-219	604.4	-398.3	-3011.9	-165.8	611	-3859.3	549.1	-468.7
	12	-54.6	378.4	259.5	-6.5	291	215.1	225	-25.2	51.5	94.8	-151.8	-382.5
	24	-18.1	-1051.7	-4932.4	174.7								
27	0	-7985.6	-471.4	536.2	-173.5	565.7	67.8	802.8	-1040.1	832.5	-106.7	-361.2	453.6
	12	40.1	-622.4	-140.1	-63.5	383	-80.7	-149.3	-162.5	-72.1	84.6	-199.5	84.3
	24	-575.8	-136	87.8	-713.6								
28	0	350.8	-1145.1	-210.1	1038.6	-18.8	-1098	210.8	210.4	-802.5	-696.9	212.9	381.2
	12	-524.5	73.2	-33.8	8192	130.9	-435.1	-880.7	-549.6	133	-97.3	76.1	177.6
	24	-12	261	-168.5	177.8								
29	0	-3820.4	4929	586.4	357.3	208.7	-1327.9	525.2	-426.1	-290	-84.7	-283.1	-992.5
	12	-777	727.9	522.5	-130.6	-31.9	58.6	655.4	-233.7	-2.1	1969.1	-24.5	122.3
	24	-1434.3	-529.6	-127.7	399.8								
30	0	-253.6	580.2	3009.4	2563.5	-2683.8	34.2	-168.6	349	322.5	-399.6	626.4	-743.8
	12	-938.8	151.5	8.4	-310.6	199.1	116.5	181.6	-192.6	-90	93.5	178.1	-6795.7
	24	77.2	689.1	269.7	216.1								
31	0	-74.9	2728.3	143	1142.2	181.9	3492.9	5593.3	631	-296.2	-136.5	464.6	-234.4
	12	639.1	-401.2	206.2	-0.4	244.2	-132.8	323.2	-611.7	-270.7	-438	756.8	194.9
	24	30.6	-370.5	-55.2	-210.9								

Table 4.A.25 – Interleave VQ codebook 0 for scalable coder (WINDOW TYPE : SHORT)

filename	contents	mode	number of elements	number of vectors
scmdct_2	MDCT	enhance SHORT	24	32

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	613	-1918.4	-968.9	892.6	419	79.3	-77.8	-240.8	162.1	493.8	-1223.7	720
	12	181.3	-1005.3	-10.1	1160.7	464.2	-1110.3	138.9	-7755.7	-603.2	-621	779.9	-144.7
1	0	199.2	449.8	-147.5	-650.1	474.8	-258.1	635.7	-975.8	768.7	-1061.6	606.4	-437.7
	12	-535.1	214.6	-934.9	-8192	-104.9	84	-393.5	-261.4	26.9	1021.8	-110.7	-818
2	0	-2191.5	516.5	-512.3	-7841.8	-1863.1	930.8	296.5	-854.1	321.5	-104.4	745.6	1428.2
	12	-71.9	-406.6	549.5	1065.5	1224.3	-899.9	-418.3	4	-588.7	125.3	-942.3	-1317.2
3	0	-5741.3	1175.1	-3780.8	-1979.1	3997.5	2363.4	1948.9	-1075.8	270.2	1113.4	-1514.8	36.5
	12	-215.7	1244.3	1454.6	-629.9	526.6	-31.9	960.9	-292.7	1256.4	1770.9	-1638.3	-1141.1
4	0	1869.4	1185.6	515.5	-381	-1273	-942.5	-988.9	-595.9	-954	-663.4	-547.1	731.1
	12	-360.1	-93.3	7036.1	-1416	1070.4	-2318.2	314.9	440.7	72.7	-60.8	-1731.8	1086.1
5	0	-535.5	-152.7	-523.3	1189.4	100.9	-560.4	-269.2	-1528.7	-482	-672.3	204.8	-1138.3
	12	-85.9	-914.3	-266.2	1683.5	-411.3	-522.6	427.8	436.3	166.2	8119	-1297.1	173.8
6	0	1996.2	1353.1	2318.5	4260	-3061.5	2192.3	702	-2385.7	1653.8	1494.3	-1128.3	807.4
	12	-1586.9	1139.7	-1618.1	217.8	-808.5	827	-756.3	1528	493.9	-1344.4	-4055.4	-118.7
7	0	164	620.7	-359.6	201.5	-932.1	8192	-101.1	-1723.8	-94.3	-950.4	198	903.7
	12	-602.2	-198.7	-514.4	1396.4	-756.6	-429	370.1	-249.4	-59.5	221.6	-1645.2	1282.2

8	0	-959.2	-2799.8	2031.5	1081.8	3374.3	3582.4	603.2	777.2	-1314	2646.5	340	-1727.3
	12	1645.3	173.9	-558.3	-807.2	-645.4	-2908.5	-525.6	2070	269.2	-86.7	3020.5	-6239.7
9	0	-474	-489.6	511.5	-707.2	247.4	1316.2	-45.1	-823.5	-8016.4	-1546.4	236.2	-143.7
	12	143.7	-385.4	-375.3	-1493.8	716.5	334.3	-156.9	658.5	-48.5	-1005.8	-2546.6	320.7
10	0	896.7	-1018.5	-8192	-1013.5	-380.1	657.3	94.6	-263.9	421.1	-783.9	-149	-427.7
	12	555.6	264.4	393.1	-322.4	255.5	-175	-628.9	673.4	1279.4	-641	-1016.1	-1884.3
11	0	-513.2	-335.3	-37.1	-805.8	1171.8	262.9	-1791.4	-8099.1	815.1	-783.7	296.2	-601.5
	12	-192	86.9	-3.9	974	2015.1	161.7	-1157	-490.5	-1972.9	-985.8	381.7	525.7
12	0	-2340.4	-5661.8	4040	-2018	620.7	-1192.3	82.4	-2415	-531.9	-408.1	1410.8	-124.5
	12	1669.3	1686	351.5	-150.9	-46.5	775.2	-550	-189.9	1333.1	33.9	-1745.5	-417.2
13	0	65	22.3	770.3	-610	1222.8	625.5	923.8	611.1	778.8	-72.4	-793.9	-8138.9
	12	-652.8	-443.5	-326.3	372	-835.5	-932.2	-438	636.3	552.6	-645.1	1783.7	-1578.1
14	0	-1400	-98.5	3057	3613.4	2282.3	2572.4	3484.4	-3571.9	481.7	-3392.1	1297	1167.4
	12	161	-1064.2	-313.3	1599.9	-6.9	470.8	653.7	136.3	2168.9	1038.2	856.7	363
15	0	-785.2	-384.5	-166.7	305.1	-465.8	-565.9	-1368.1	-287.8	52.2	207.1	2756.1	487.7
	12	422.8	-1724.3	-646.7	219.3	52.2	-7309.5	-398.8	871.1	614.9	-965.9	-94.9	-1695.4
16	0	3931	-7212.4	-30.7	-853.7	-141.4	-138.3	-196	295.3	-78.4	364	388.5	-1395.2
	12	-1237	1038.9	703.5	87.8	546.8	-9.3	-696.1	-777.8	-78.2	-475.6	224	-268.8
17	0	2059	2976.1	2990.4	-3869.7	-1417	1535.6	-2343.5	-983.8	-969.7	1746.4	149.8	2876.3
	12	-1391.5	-262.8	-1512.9	980.8	-1948.6	-752.1	-564.2	1323.3	1.1	-518.9	2387.7	617.5
18	0	263.7	78.7	663.8	-310.6	-2441.4	-1218.3	-481.4	1743.2	514.8	-483.3	1094.8	655.3
	12	269.5	146.2	-913	-744.6	7872.2	2233.3	401.5	948.2	-1182.8	1871.3	459.9	-672.7
19	0	23.6	461.8	754.2	-411.9	451.3	245.3	-8192	-294.2	-192.5	-819.3	-791.4	-1601.9
	12	-430.8	-131.7	573.7	1347.4	-345.2	46.1	1877.7	-798.3	-42.5	-84.7	96	1252.6
20	0	381.9	3174.6	-608.4	-2156.8	-3019.3	781	4767.2	-1575.8	920.3	-660.6	-779.4	-2567.7
	12	1139.5	647.3	98.2	2453.2	-1062.3	-608.9	-1438.9	1269.2	-1554.3	-979.2	-244	1879.8
21	0	-502.3	-843.6	631.9	843.7	708.5	-992	-471.3	-1323.4	788.6	-952.4	833.7	1104.1
	12	555.9	27.4	1181.2	-183.5	291	803.2	-106.9	-517	-504.5	108.5	8192	2984.5
22	0	433.6	246	-1589.2	-817.6	-708.4	-1522.4	-1661.6	1227.3	-1094.2	-491.1	237	-887.9
	12	1532.3	1569.8	-299.8	-219.7	-517.8	407.6	414.2	122.3	7957.8	-501.4	-406	202
23	0	-4188.3	247.2	437.9	1234.2	-3749.7	-5089.5	-159.7	-401	1524.7	-1408	-3081.6	-60.8
	12	1318.4	-63	-1692.2	-71	-575.4	-1530.1	-6.8	838.7	-913.4	-431.7	1032.4	-2078.2
24	0	656.5	1578.5	-2239.9	764.9	1307.3	-4063	2133.6	-4375.1	-2720.8	-82.3	-4.4	1379.8
	12	1452.4	-794.8	-1084.7	-720.3	-1685.5	-105.5	1770.7	1205.6	-635.2	799.4	1327.4	-776.7
25	0	406	353.4	1207.9	100.3	-554.4	-363.5	812.3	-219.5	368.2	-818	-1283.4	115.5
	12	949.7	-348.7	864.5	-336.3	326.9	-779.5	7874.7	-142.3	1076.4	-848	-246.6	-3475.1
26	0	-8192	-3.9	-103.4	-300.1	-451.7	-318.6	197.6	139.9	-167.5	-386.7	1758.9	755.7
	12	75.4	-465.2	329.6	557.2	-219	526.8	896.1	435.7	-509.5	65.4	-407.4	-1614.7
27	0	246.7	-2.2	-202	763.1	-8192	985.9	776.4	51.7	74.6	180.7	-360	463.2
	12	226	-83.2	1488.2	-1249.9	-1186.7	167.1	-211.1	-562.4	75.1	1007.6	400.6	-2326.2
28	0	336	441.6	215.6	104.9	-216.9	860.4	-1256.2	505.1	596.4	-7828	542.6	-218.9
	12	-1139.2	175.8	-1623	383.9	66.5	-230.2	-732.1	505.2	417.1	587	210.5	-1437
29	0	572.1	-128	823	-312.2	59.7	100.7	-720.4	695.3	-790.7	-1235.8	891.5	1663.5
	12	347.7	7774.4	-121.7	-516.7	-906.6	929.5	-315.7	-397.6	-1038.2	-1197.8	-92.7	-929
30	0	-1050.2	-1276.6	-428.5	-141.9	88.4	-1400.9	34.3	763.6	-203.7	-854.5	-713.6	13.7
	12	-7944.3	265.6	-520.4	-532.2	1207.4	-1606.8	480.2	-31.7	-2769.1	-386.9	-52.8	-45.7
31	0	-4196.5	-786.1	-474.8	2859.1	-3151.7	3652.5	-2296	1179.6	-1717.4	1482.6	-2578	-401.3
	12	1222.3	522.7	124.4	2285.2	-180	-849.8	-180.3	1514.7	-1489.2	-946	1162.9	1043.9

Table 4.A.26 – Interleave VQ codebook 1 for scalable coder (WINDOW TYPE : SHORT)

filename	contents	mode	number of elements	number of vectors
scmdct_3	MDCT	enhance SHORT	28	32

VN	EN	0	1	2	3	4	5	6	7	8	9	10	11
0	0	-1587.4	-8151.2	466.2	476.5	1310.7	-36.4	-835.2	-385.1	-329.8	-983.2	-724.4	-861.8
	12	-354.7	-992.7	197.9	554.3	-77.3	207.1	-977.7	-660	-331.6	602.2	706.1	-1584.4

1	0	-495	61.2	-263.6	1248.6	-1573.1	-713.3	2527.9	-6633.3	-696.5	247.3	1391.3	-289.7
	12	-324.9	489	-1072	-459.6	-211.1	-736	385.2	-1296	-37.3	1785.9	-258.4	-459
2	0	433	-524.1	264.9	-41.6	7135.3	1978.2	105.5	-52.2	-396.5	-306.8	-1282.4	412.9
	12	-21.3	1616.7	-46.3	-844.6	-1902.3	1368.5	-133.6	211.3	1114.3	457.7	2710.2	-1426.3
3	0	1621.1	-382.5	941.6	1426.9	-4604.8	2534.3	-1221.5	267.8	-4996.4	-1924.1	763.4	1417.4
	12	254.9	-620.3	325.2	358.6	669.7	-799.6	529.5	-319.1	-364	-423.1	-2416.2	-1041
4	0	-357.9	860.6	339.1	143.7	-188.7	1506.9	-903.6	-898.6	6330.3	-3820.3	829.1	-11.5
	12	220.8	-477.8	1134.2	253	383	-1115.3	-1149.9	848.9	-1113.5	107.8	-868.4	295
5	0	-8192	-591.2	-299.6	524.8	1471.7	73.1	-869.7	-211.1	-162.2	139.5	-1121.5	-1009.6
	12	91.9	160	-270.5	67.1	-275.7	-861.7	-722.4	-488.8	-623.8	698.3	697.8	-264.1
6	0	-818.4	786	-608.1	-61.4	1509.7	7783.9	-37.6	2065.4	-638.7	198.9	35.8	-824.1
	12	791.2	739	605.3	-1152.7	-372.9	424.6	1346.8	-18.8	-625.8	604.6	-384.4	284.6
7	0	3029.7	-1750.6	4506.6	-870	513.2	2301.4	927.1	2530.7	228.1	-2474.1	1344.5	-78.5
	12	153.2	-1658.2	-537.2	-280.5	642.4	2369.1	500.4	-584.8	-170.5	-303.1	1704	136.9
8	0	-1569.8	-1288.6	201.5	-1157.1	-466.5	797.8	371.5	-1819.5	537.5	131.8	-3898.4	5382.6
	12	1775.5	-542.9	-1048.7	-370.1	-635.7	-952.7	-997.7	-41.5	726	11.2	-81.3	13.6
9	0	-2450	1788.6	-1971.3	614.4	-787.8	-345.9	-1830.9	4736.7	154.9	76.6	1477.9	576.4
	12	1549.4	-873.6	-1628.5	-5.8	-1817.4	1087.2	-119.7	-4009.5	-362.6	-2510	-3032	-1122.4
10	0	1489.3	5731.5	4239.4	396.9	1859.3	-1909.5	302	-790.5	1461.5	-1084	-510	-1498.7
	12	788.4	-259	965	-536.7	428.1	622.7	-227.5	-1693.9	-98.6	217.7	-852.7	-1826.4
11	0	727.1	-471.3	-3938	4596.7	155.4	2508.6	-1160.6	-959.7	1337.4	-1124.4	782	747.3
	12	169.4	-577.7	2942.3	-638.7	1307.7	1236.2	-449.5	-649.7	294.4	1321.4	-699.9	364.8
12	0	-3489.7	-557.6	1321	-1750.9	-324.9	-704.9	5375.6	2709.4	343.7	-3100.4	-1881.5	172.9
	12	-1917	847.7	679.8	-268.1	974.4	27.3	-60.4	-1128.4	12.1	-2983.5	-354.1	945
13	0	254.3	1387.7	1209.8	598.9	277.1	-934.4	-7093.7	1923.9	1015.7	-357.3	-390.3	1251.3
	12	-946.7	480	208.4	1134.5	750	-716.6	-726.8	794.7	-336	75.3	807	-2576.3
14	0	1188.4	-248.5	718	-1366.1	-3130.4	-1170.6	-101.4	1117.5	1004.6	7.1	-1020.9	63.7
	12	-483.1	-1065	878.5	-1709.5	-6818.4	1088.2	363.8	-686.1	-1432.5	-405.5	343.5	-391.3
15	0	-945.3	2896.2	362.4	-2761.9	2342.6	805.7	-619.8	490.8	-2910	-798.1	-59.8	409.3
	12	-1168.1	149.5	550.7	4936.1	-563.9	-328.6	-320	616	1443.7	381.4	-1751.2	-738.3
16	0	1408.8	-324.5	-250.1	-42.1	-718.3	654.3	608	324.9	-863.8	596.2	-2048.1	-1469.7
	12	1793.5	-211.2	-15.2	394.4	992.8	-6976.7	1115.8	-1024.9	823.3	216.9	-1298.7	2454
17	0	-2586.3	-2557.9	-1035.1	-2009.8	-1809.3	-5664.7	-661.2	-987.2	-261.8	-731.4	806.3	972.9
	12	-2113.7	1892	1078.4	1751.2	-443.4	1183.3	1097.3	570.2	-57.2	-956	-1259.4	53
18	0	1515.5	941.4	-381.6	1674.6	-81.4	121.6	2131.6	-556.5	1081.9	1872.4	-816.3	-1076.9
	12	496.7	7226.8	-416.5	-382.9	508.1	1137.3	-1334.6	790.6	1024.6	613.2	-441.6	-1065.9
19	0	-406.3	380.5	681	-1482.5	-2079.6	-1901.1	-3583.9	-3315.9	-447.2	-63.2	-691	-1701.2
	12	578.3	-2632.7	663.9	30.6	1611.2	3718.7	867.1	1683.5	50	-1270.9	-1571.8	-1539.7
20	0	321.7	-618.9	1318.2	-2858.6	-95.2	109.2	1145.3	1812	-278.3	2607.3	-359.3	1530.9
	12	-1664.7	678	2133.4	-1815.8	1561.3	56.1	-134.9	-12.2	-342.6	6635.7	2642.9	2408.3
21	0	757.9	-357.8	798.2	492.1	-1311.4	742.9	4154.7	105.9	571.6	2169.7	903.3	334.5
	12	-3191.2	-1928.7	-1051.4	1526.9	245.2	-1750.8	-1104.9	1049.2	5111.9	-1193.6	-521.4	-734
22	0	-1838.4	1235.6	1095.1	2725.7	2497.5	-1924.5	1151.5	603.4	-1416.1	2696.4	-268.5	-1621.3
	12	399	-529.9	2481.5	-123.4	2178.4	857.2	-991.2	3837	-385.4	-212.5	-3028.1	4468.3
23	0	1330.4	2003.6	-5625.7	-1541.8	709.1	-203.4	722.3	468.9	1065.2	1010.1	1321.9	653.3
	12	-1503.7	-890.1	126.5	595.1	1292	-120.9	867.1	309	-1041.5	-1233.9	4458.3	1747.7
24	0	3786.4	166.8	659.1	-7165.8	736.4	760.8	-157.3	-832.6	-726.2	-558.5	-1335.9	-1448.4
	12	1227.1	310.8	-1051.7	-528	-791.4	629.3	33.1	-1012.9	-632.4	-129.1	-499.5	1025.3
25	0	-1451.7	246.9	244.9	149.8	5.9	156.5	-51.5	-109.4	478.2	306.5	7042.5	253.4
	12	122.7	110.6	-2.7	-947.8	-1074.1	318.6	81.2	44.4	-1188.6	980.2	-1001.7	1166.1
26	0	-1751.7	1177.9	-1175.2	-2322.5	-4825.7	1590.5	258	229.3	-92	167.1	71.5	-2142.2
	12	-2665.3	1556.1	-2104.4	374.6	1627.4	-847.9	1737.1	110.3	594.9	1048.1	-177.5	-207.9
27	0	121	-145.1	-561.2	420.7	-152.6	-871.5	-1213.1	-294.2	-868.6	807.8	-220.4	981.4
	12	89.8	-313.7	-7218.6	424.1	-315.4	-975.3	-199.4	-893.3	283.3	-108.3	775.2	785.3
28	0	2464.9	-3039.7	-673.7	188.1	1483.1	1089.5	-446.4	283	648.4	2184.9	-1143.1	576.4
	12	1052.9	281	-2367.4	1505.7	-702.3	628.9	4176.2	1901.7	-1451.3	-460	-4274.9	174.8
29	0	9.6	362.4	362.1	802.5	497.8	-914.2	-1697.2	-45.5	-1122.9	648.9	720.2	916.9

	12	-371.9	588.3	162.3	-6838.7	-849.4	-560.4	316.3	180.4	-229.1	-2966.6	-638.9	437.5
30	0	3819.9	685.6	-662.9	1849.7	1157.1	-2943.1	2673	-153.4	-1106	-1384.2	869.4	1226.8
	12	-513.6	-352	1064.3	4032.2	375.9	-824.9	-1518.7	1262	-2905.1	160	-315.9	-3058
31	0	-125.4	-618.8	379.6	-840.3	-975.8	-1430.1	1370.9	583	-452.9	-73	-374.4	-1129
	12	7405	175.4	123.6	-458.8	1692.4	-318.4	-1535.9	521.6	-277.1	-690.6	899.4	3196.9

Table 4.A.27 – Bark scale codebook

Filename	contents	mode	number of elements	number of vectors
Fcdl	bark envelope	core/enhance	6	64

VN	0	1	2	3	4	5
0	2.59553	1.59708	-0.21402	-0.17514	-0.09628	-0.26982
1	-0.31846	-0.28922	-0.32107	1.15714	1.03967	-0.03709
2	2.80202	-0.48079	2.15678	-0.37126	-0.36071	-0.16559
3	0.19375	-0.32843	-0.30811	0.27793	-0.2106	-0.07445
4	2.2074	-0.43946	-0.3363	1.40869	0.0118	0.04888
5	-0.50163	-0.3072	-0.34648	-0.36553	0.08881	0.66861
6	-0.5478	3.10285	2.923	-0.42525	-0.30431	-0.12124
7	0.36265	-0.59101	-0.40694	-0.37653	-0.02823	8
8	0.6108	-0.2944	0.84234	0.50441	0.15898	-0.19708
9	-0.26828	-0.42775	-0.55847	8	0.26906	0.09807
10	0.58163	0.69724	-0.38205	-0.32305	-0.04204	-0.18015
11	-0.1531	-0.52465	-0.5112	-0.46322	8	0.60858
12	8	-0.37894	-0.41379	-0.57177	-0.46141	-0.54807
13	-0.3409	-0.46601	-0.40139	2.77429	-0.30628	-0.29932
14	-0.00875	1.00366	0.35027	-0.14719	-0.12343	-0.01017
15	-0.42826	-0.25287	0.5889	0.00455	-0.18978	0.72595
16	-0.41718	-0.42836	0.44293	-0.26096	0.67149	-0.15131
17	-0.48094	0.40551	0.63479	0.10576	0.60523	0.00965
18	0.5875	0.17574	0.22913	-0.07386	-0.25141	-0.28609
19	1.54175	-0.38932	-0.26886	-0.37004	1.26474	-0.04734
20	-0.30997	-0.45256	2.71793	-0.29703	-0.34868	-0.27166
21	0.80891	-0.39427	-0.36544	1.05681	-0.26354	-0.07337
22	1.69802	0.38677	-0.19213	0.04237	-0.07601	-0.16281
23	0.82683	0.70298	1.21216	-0.23461	-0.02006	-0.24785
24	-0.4438	-0.46604	-0.44381	-0.40097	1.97582	0.05232
25	0.71883	0.61561	-0.1844	0.68805	0.00106	-0.01607
26	-0.38766	8	-0.06823	-0.22391	-0.52152	-0.42881
27	-0.4812	-0.4593	-0.4612	1.54307	-0.31462	0.33229
28	0.72907	-0.31141	-0.38468	-0.36775	-0.3587	0.7
29	0.88654	2.10076	-0.1964	-0.23175	-0.15172	-0.15909
30	-0.51999	-0.53691	-0.52863	-0.4753	0.54786	-0.32601
31	-0.4771	2.06467	1.01919	-0.28684	-0.14439	-0.05744
32	-0.57224	0.42109	0.5242	-0.2856	-0.1884	-0.15752
33	-0.24585	-0.44169	8	-0.43055	-0.40749	-0.43133
34	0.20221	-0.25994	-0.31787	-0.30064	0.6737	0.05431
35	0.32692	-0.4254	-0.45128	-0.45458	-0.16592	-0.3783
36	1.42145	-0.42477	-0.43237	-0.33452	-0.32365	-0.35176
37	-0.52165	-0.47377	0.32662	-0.34749	-0.24129	-0.18172
38	-0.46545	0.29674	-0.31146	0.41441	-0.12466	-0.07952
39	-0.26585	0.07241	-0.26056	-0.2953	-0.24943	-0.22879
40	0.34838	-0.3777	0.48323	-0.31334	-0.0997	-0.15367
41	-0.36403	-0.40947	1.43126	-0.48284	-0.33652	0.15962
42	-0.50425	-0.42948	-0.31036	0.28637	0.32368	0.02942
43	-0.44723	2.90793	-0.35379	-0.37063	-0.16951	-0.00506
44	-0.41686	-0.31531	0.3233	0.63998	-0.13565	-0.25397

45	-0.57783	-0.55613	-0.52567	-0.35549	-0.35857	0.13331
46	-0.5269	2.92107	-0.39287	2.50676	0.83325	-0.34838
47	-0.35037	1.58353	-0.14216	-0.14513	1.25833	0.02395
48	-0.54618	-0.51853	-0.5225	0.6915	-0.34809	-0.32408
49	0.91424	-0.39755	-0.23708	0.06298	0.32884	-0.27221
50	-0.26031	-0.34193	1.87043	-0.21861	1.45874	-0.0149
51	2.71047	-0.44695	-0.35421	-0.35093	-0.21284	0.18185
52	-0.54924	-0.36134	2.97081	2.32648	0.01315	0.14583
53	-0.51477	-0.40314	1.23469	0.2071	0.00622	-0.15786
54	-0.37564	0.82963	0.77552	0.72145	-0.21951	-0.21676
55	1.59584	-0.35791	0.89497	-0.26614	-0.12674	-0.13373
56	-0.42762	1.49492	-0.31584	0.6556	-0.06384	-0.07354
57	-0.38684	0.71379	-0.43055	-0.24995	-0.29525	0.42418
58	-0.55264	-0.52329	-0.57869	-0.60007	-0.61091	-0.56922
59	-0.46465	0.55867	-0.28994	-0.14188	0.53955	-0.1124
60	-0.40785	-0.46448	1.26074	1.48856	0.10128	-0.10592
61	-0.427	0.70964	-0.30237	1.50654	0.0617	-0.27741
62	-0.43966	1.39518	-0.39417	-0.44686	-0.35716	-0.39151
63	-0.43383	0.9626	1.83882	-0.37505	-0.235	-0.05237

Table 4.A.28 – Periodic Peak Component codebook

name	contents	mode	number of elements	number of vectors
pcdl	PPC	core LONG	10	64+64

VN	0	1	2	3	4	5	6	7	8	9
0	1.28888	0.26874	3.16758	1.37315	0.49535	0.94071	1.44487	0.37004	-0.70932	-0.05462
1	-0.02234	-0.20001	0.19604	0.26626	0.75466	-0.81913	0.03708	-0.29647	-1.78632	-0.18008
2	-2.19075	0.13333	2.93822	-0.22869	0.54111	0.51598	0.29629	-1.60571	0.48978	-0.40555
3	-0.16435	-1.43562	0.12102	0.1109	0.83013	0.16508	-0.07873	2.78031	0.15479	-0.28546
4	3.06554	-1.41708	-1.88582	0.14353	-1.66046	0.26375	0.08646	-0.41044	-0.23005	0.47026
5	-0.11899	-0.14149	0.03797	0.01927	3.72378	0.16278	0.28207	0.15989	-0.17563	0.48802
6	-0.19166	0.05232	0.68887	0.24894	-0.48014	0.01794	-0.09456	-0.03353	-3.71716	-0.37879
7	-0.08722	0.69099	0.52234	0.74945	-0.62316	0.58827	0.30028	1.0085	0.24395	0.06055
8	0.12958	-0.22199	0.96387	3.13907	-1.96687	0.53834	-0.69258	-0.04179	-0.39869	-0.03142
9	-0.02278	0.45226	-0.46875	0.03839	2.35515	-0.02039	-0.03662	-0.33315	-0.3716	-3.22201
10	1.50588	-0.78015	1.43802	-0.42078	-1.56427	-0.32711	-1.81516	-0.1517	-1.39942	-0.70676
11	0.03549	0.97029	0.17778	2.40233	-0.1207	1.79612	-0.09612	-3.32378	-0.06238	-0.47122
12	0.11414	-0.06499	1.95345	-0.03737	-0.08911	1.86812	0.03298	0.3419	-0.9445	0.26631
13	-1.20479	-0.70412	0.30736	1.66473	0.25032	-1.62769	0.26336	-2.39109	1.66663	-1.01706
14	0.27057	0.28062	4.60481	0.14478	-0.04332	-0.26043	-0.15716	-0.52842	1.15402	0.02068
15	-0.02468	-2.63793	-0.06035	2.17716	0.55198	1.33668	0.18469	0.73687	-1.59535	0.47395
16	0.08763	-1.03259	0.29432	0.23395	-0.21428	0.13532	-4.58495	0.5589	1.19783	-1.8095
17	-1.87625	1.93644	0.2355	0.60058	1.07984	-0.13744	-0.89547	-0.8501	0.80741	-0.42413
18	0.23718	-1.82628	2.62266	0.48302	0.5773	0.00497	-0.88499	0.11386	0.50195	-1.61181
19	2.29405	0.76792	0.77194	0.45336	-0.15374	-1.68579	0.40538	-0.73821	-0.52721	0.27069
20	0.31311	0.67546	0.05131	-0.53788	0.75649	-1.92602	0.44063	-3.20197	0.24121	-1.52231
21	0.38234	-0.34155	-2.34677	0.44454	-0.12667	-0.01386	0.83684	-3.42385	0.27861	-0.62057
22	-2.294	-0.00411	0.10751	-2.0974	0.55445	-1.90704	0.46176	-0.22817	-1.05712	-0.31803
23	0.08838	0.104	0.07782	0.28709	0.77533	0.5377	-3.49315	-0.15122	-0.71822	-0.55433
24	-0.12964	-0.15924	0.08426	4.31671	0.35474	-0.3812	0.08329	-0.5072	0.22773	0.10575
25	0.37323	0.11307	-0.26726	1.19933	-0.2052	3.97109	0.23754	0.42287	-0.17526	1.6363
26	-0.8932	0.98569	-0.17186	-0.66099	0.34743	0.31512	0.56153	0.03641	-0.2279	-0.32669
27	0.49296	1.50705	0.11747	3.64671	-0.08886	0.95113	-0.10827	0.24228	-0.2188	0.09239
28	-0.05516	-0.36735	0.06542	-0.10454	-0.64008	-0.07052	-0.50515	2.55652	-0.94717	-1.30125
29	0.09405	0.74533	-1.13331	0.37229	-0.38024	0.17472	0.61134	0.20712	-0.10988	-0.49257
30	-0.11187	-0.17222	-1.73824	-0.13261	-0.52195	0.62213	-0.08559	0.17988	3.15089	-0.24704

31	0.01548	3.74936	0.08098	-0.20425	-0.05489	-0.31726	0.69418	0.32569	-0.1634	0.50865
32	-0.17257	0.28592	2.00649	0.53473	2.57991	0.33445	-0.33589	-0.71256	0.51466	2.46521
33	-0.09483	1.08439	0.03998	0.11961	1.85047	0.00631	-0.45111	-0.71269	0.15621	1.16161
34	-0.02632	2.00056	-0.32356	-0.56487	0.10923	1.26427	-0.2674	2.41749	-0.16922	-0.8848
35	-0.0544	0.16837	2.56225	0.1894	-0.22447	-0.39572	0.49943	0.11457	0.35493	0.89878
36	-0.26616	-0.43364	0.3558	0.05177	0.50149	-0.24278	0.17047	-0.12116	5.45574	-0.15218
37	0.06203	0.23433	-0.99429	0.12958	-0.628	-3.51385	-0.24958	0.06293	-0.89521	0.2426
38	3.72107	1.23981	-0.88335	0.46989	-0.3001	-0.42813	0.41867	1.0639	0.31391	-0.57627
39	-1.12063	0.44526	1.91996	0.34752	-0.524	-0.6746	-2.59639	0.11581	-0.20243	0.28307
40	-0.03832	-0.01316	-0.03846	-0.007	0.04931	-0.09357	0.06964	-0.00049	0.21041	-0.08892
41	0.10125	-0.03012	-0.03539	-2.37631	0.28491	2.4899	-0.34378	-0.73052	0.52019	-0.10068
42	1.15802	0.7305	0.86786	-0.15069	2.3452	-0.69111	0.57534	1.29397	0.15267	-0.16541
43	-0.12218	-0.03125	-0.01363	-1.5446	-0.27404	-0.62664	0.46486	-0.44052	0.17302	0.77978
44	-0.03625	-0.33463	-1.59515	-0.42669	-0.3637	3.45674	0.35785	0.34057	0.26874	-0.66518
45	0.12284	0.01668	-0.36273	0.00005	0.05147	-0.29728	-0.00085	5.10734	0.41718	-0.05075
46	-0.20474	1.11663	-1.4853	-1.6009	-0.02503	-0.24016	-0.09167	2.05302	0.41858	-0.52989
47	-1.22133	-0.80464	3.2915	0.29576	-1.01545	0.85362	0.25387	1.69078	0.05201	-0.3414
48	0.10775	-0.30053	-0.19873	0.20553	0.38424	0.2795	0.33175	0.10667	-0.43378	5.0211
49	0.06324	-1.1164	-0.34474	-0.03852	2.47664	0.30992	0.34303	-3.0838	-0.26944	-0.15003
50	-3.87998	0.08673	-0.92651	-0.28277	0.7912	0.06152	0.05675	-0.04378	-0.15055	-0.49744
51	0.06965	-0.24747	-0.30479	0.44796	-0.38039	0.30899	-2.49186	0.15352	-0.07609	2.77728
52	0.03771	-0.07037	1.44405	-0.08627	0.01465	0.65797	-0.20831	-0.02516	3.82295	0.58679
53	-0.28911	0.00076	2.25478	-2.67418	1.08246	0.20249	0.62566	-0.04001	-0.20703	-0.15767
54	1.76926	0.21839	0.4214	-0.18353	-0.23904	0.25714	-0.23697	0.07767	0.53228	0.22643
55	0.19622	-0.064	0.10897	0.79608	-1.20257	-1.10434	0.30441	0.49781	3.11769	-0.29082
56	-0.31973	-1.36087	1.1628	0.16043	-1.01046	2.64181	-0.19644	-0.27073	-0.77847	-0.42009
57	0.04065	-2.4468	-0.45823	0.51819	0.08004	-1.72072	-0.52	0.59359	0.37168	0.78352
58	-2.06255	0.4418	-1.32138	2.39634	-0.89775	1.60514	-1.52266	-0.22682	-0.10606	-0.5798
59	-3.31681	0.85384	-0.27833	0.29965	-1.55139	-0.12387	0.08035	-0.41261	0.26349	-0.21486
60	-0.91388	-0.98165	-1.13409	-1.47777	0.23383	1.1217	-0.27391	-2.83883	0.37536	-0.83449
61	-0.30317	0.03241	-2.27905	0.39328	-0.10539	2.62031	-0.21607	-0.11414	-3.2875	0.16852
62	0.05213	2.18937	0.84015	1.33077	-0.05358	1.3409	-0.09204	0.09985	0.93062	0.59825
63	0.12392	1.59708	0.9337	-0.90451	-0.12543	-2.01707	0.37513	1.94555	0.23099	-0.70055
64	2.13905	0.10191	1.24373	0.51358	1.63431	-0.08351	0.30739	-1.16675	-0.35279	-0.37096
65	0.0812	1.07058	0.15907	-0.09428	-0.14151	-0.4618	-0.21325	0.45456	0.66027	3.37816
66	-1.32206	-0.4821	1.43573	1.74563	2.01948	-1.56545	0.17182	0.45271	0.0628	-0.91456
67	-2.3898	-2.35197	-0.22554	0.33998	-0.79031	0.06321	-0.01377	1.07176	1.19673	0.2861
68	1.24864	-0.63868	-1.0763	-0.28812	1.14537	1.80661	-0.15262	2.16631	1.67133	-0.29882
69	-0.05193	-1.29291	-0.85556	-0.03597	4.35258	-0.32884	0.79375	0.41554	0.35156	-0.57719
70	-0.1198	-0.05545	0.43083	-0.10895	-2.37948	-0.49303	2.75568	-0.04768	-2.33369	-0.07613
71	0.21686	1.41341	-0.42804	-2.70242	0.61944	-0.06463	-0.43017	-0.1903	-2.0755	-0.07512
72	-0.02184	0.4818	0.88869	4.23426	0.32367	-0.16813	-0.0971	-0.93254	0.52898	0.05603
73	-0.15532	0.52294	-1.60824	-0.18328	3.23327	0.1135	-1.45172	-0.0493	-0.23632	-0.25873
74	-0.05867	-2.59416	0.8312	0.31796	-0.30826	0.2373	2.77116	0.02446	0.55313	-0.22707
75	-0.51455	1.30626	-0.77779	2.71737	0.5061	-1.37105	0.15878	0.48821	0.25587	1.37404
76	-0.12972	0.16455	2.0636	0.25143	0.28128	2.95291	0.08582	0.09976	-0.25123	-0.30165
77	-0.04632	-1.3975	-0.14369	0.86661	0.47791	-3.43773	-0.08283	0.40818	-0.40507	1.10291
78	-0.0182	-0.15582	1.34311	0.06751	-0.14121	-0.53351	0.03831	0.5043	2.10788	-0.58678
79	0.39477	-3.04864	-0.45931	2.01978	0.04376	-0.52536	-0.31599	-0.12492	-0.10121	0.11121
80	-0.03452	-0.03057	-0.09894	0.03226	0.01192	-0.48283	-4.65939	-0.11003	-0.3613	0.079
81	-1.31754	0.0574	-0.2971	0.08453	-0.30721	0.86434	2.45968	2.51238	1.23547	-0.50231
82	1.50294	-0.24216	2.28121	-0.90775	1.04895	0.21859	-2.06629	0.08329	0.34822	0.1581
83	0.36466	0.70677	-0.918	2.47546	3.79165	-0.24031	-0.37853	-0.32197	-0.37636	-0.02901
84	-0.12627	-0.04525	-0.82218	-2.26543	-0.84347	-1.03939	0.26653	-2.47079	1.23837	0.75649
85	0.24064	-0.62543	-2.94637	0.47164	1.58428	0.23203	-0.59003	0.3325	-0.13853	0.14422
86	-0.17463	-0.12583	0.85598	-0.17722	1.54264	0.05216	-0.08885	0.88151	0.46255	-0.48946
87	-0.06676	0.36942	0.58086	0.11589	-1.18536	-0.12857	-2.80603	0.00658	1.06034	-0.58457

88	-0.1561	-0.18171	-0.63499	2.6653	0.46641	0.59787	0.30223	-0.53096	1.29943	-0.98973
89	-0.00976	1.04356	0.16697	1.99034	0.23572	2.65679	-0.00325	1.03831	-0.0701	-0.77931
90	0.13381	0.49307	0.04619	-1.68299	0.19455	0.52731	0.14125	0.99717	0.43132	-0.97925
91	-0.1563	1.8791	0.19313	0.23078	0.30029	1.5847	0.36806	-0.91846	-0.10561	2.76858
92	-0.03141	-0.27603	-1.25022	0.76867	0.37696	-1.86725	0.45545	2.35843	0.57733	-0.51505
93	0.19081	-0.02002	0.89507	-0.17124	0.07392	0.07283	-0.47206	-0.03043	-0.34827	-4.29424
94	-0.29	0.04272	-1.72027	-0.11905	0.27361	-1.60983	0.1195	-0.37867	3.50315	0.36605
95	-0.1328	4.18273	0.04841	0.61437	0.60526	-0.23885	-0.05236	-0.82974	-0.29851	0.51121
96	0.03186	-0.24892	0.25614	-0.17325	3.12521	-0.12148	0.07243	-0.3332	-0.04557	-0.77374
97	-0.02997	2.20189	0.2339	0.05403	3.47503	-0.14415	0.23471	0.75477	0.00188	0.06228
98	-1.11961	0.43678	0.61581	-3.07064	-0.02222	-0.20063	-0.19664	-0.1743	1.29354	0.31722
99	0.80714	-0.31437	3.21672	0.05728	0.42811	-0.01624	0.4669	-0.14758	0.79094	-0.49991
100	-0.04883	0.18854	-0.0422	-0.24325	0.33396	0.58096	0.07173	0.18625	4.80098	-0.04774
101	0.12902	0.04455	-0.58451	0.22529	0.18288	-3.31484	0.20874	0.1439	-2.33226	-0.00004
102	1.4006	2.53879	2.23308	-0.40397	-0.91191	0.30424	-0.59309	-0.28013	-0.53681	-0.72096
103	0.14981	-0.34273	-0.1096	-2.08103	-0.55452	1.22105	-2.32346	0.71767	-0.04973	0.88489
104	-0.06296	0.00779	0.37141	-0.07311	-0.43599	-0.69232	0.37183	0.59265	-0.20391	0.76627
105	-0.04811	0.81816	-0.06811	-3.60488	-0.30926	1.2619	-0.25378	-2.67757	0.01421	0.15661
106	-0.18735	-0.29123	-0.1443	-0.17479	-0.2036	1.97977	-0.22341	-0.17369	-2.54577	-0.41017
107	-0.15575	0.64521	-0.20741	-0.06303	2.55173	0.08849	-1.3168	-0.0852	-3.59554	-0.16002
108	0.2538	0.42032	-2.50621	0.13451	-0.06898	2.41402	0.09969	-0.09772	1.45985	0.45849
109	0.19057	-0.22542	-0.16274	-0.19255	-0.085	0.1524	-0.06344	3.54373	-0.19634	0.42005
110	-0.00046	1.3075	0.44001	0.29739	0.12153	0.45681	-0.24122	3.97823	1.01631	-0.5718
111	-1.0474	0.76204	0.79817	0.02734	-0.85256	0.27437	-0.36512	0.45476	-0.34809	-0.26607
112	0.39011	-0.04813	0.28325	-0.40521	-0.26705	0.11646	-1.00205	-0.09658	0.39184	6.38045
113	0.18221	-3.1436	0.56642	-0.41265	2.10605	0.00959	0.32737	-0.30618	-0.46143	0.62372
114	-3.41695	-0.29748	0.93423	0.2968	-0.22186	-0.04393	0.4199	-0.02121	-0.0349	0.24145
115	0.08292	-0.34821	-0.98374	0.76693	-2.30082	0.11379	-2.00866	0.10299	-1.80156	0.49759
116	0.09801	0.02398	0.87932	-0.29853	-0.23085	0.60271	0.283	0.17743	-0.0655	0.11407
117	0.0495	-2.7049	0.93988	-0.84697	-0.47967	-0.45333	-0.34552	-0.60481	0.77467	1.79495
118	2.31525	-1.35283	1.51663	0.46045	-0.94423	0.66598	0.14087	-1.23472	0.07825	0.23723
119	-0.01891	0.01146	0.05543	-0.00939	-0.25309	-0.05408	-0.08464	-0.08331	0.28796	-0.23134
120	0.14456	-1.82728	0.54162	-1.45394	0.65876	2.88044	0.04801	1.52839	0.4177	0.24366
121	0.00192	-1.51934	-0.28914	0.27953	-0.43474	0.3359	0.34038	0.09476	-0.16812	0.55648
122	1.58963	0.62054	-1.19708	0.95618	-0.19293	0.13055	-0.46467	-1.08598	1.0294	-0.00537
123	-1.63945	1.27057	1.17362	0.77949	-0.25979	1.31919	1.09939	-0.59425	-1.04889	0.33754
124	-0.36973	-1.0016	0.18249	0.95907	0.72926	2.32241	0.60848	-2.12186	-0.24255	-0.65101
125	-0.00988	-0.04139	-0.76083	0.14922	0.57784	3.97886	-0.30668	-0.34279	-0.84522	0.11366
126	-0.12785	1.334	0.11998	0.17243	-1.82888	0.49492	0.09101	-0.99747	0.39561	-1.90151
127	0.1718	1.22115	0.41649	2.03242	-0.20785	-2.41295	-0.75	1.09243	0.65831	-2.08726

4.A.5 Tables for ER BSAC

Table 4.A.29 – cband_si_type parameters

cband_si_type	max_cband_si_len	Largest cband_si		Model listed in	
		0^{th} cband	Other cband	0^{th} cband	Other cband
0	6	6	4	Table 4.A.49	Table 4.A.42
1	5	6	6	Table 4.A.49	Table 4.A.43
2	6	8	4	Table 4.A.49	Table 4.A.42
3	5	8	6	Table 4.A.49	Table 4.A.43
4	6	8	8	Table 4.A.49	Table 4.A.44
5	6	10	4	Table 4.A.49	Table 4.A.42
6	5	10	6	Table 4.A.49	Table 4.A.43
7	6	10	8	Table 4.A.49	Table 4.A.44
8	5	10	10	Table 4.A.49	Table 4.A.45
9	6	12	4	Table 4.A.49	Table 4.A.42
10	5	12	6	Table 4.A.49	Table 4.A.43
11	6	12	8	Table 4.A.49	Table 4.A.44
12	8	12	12	Table 4.A.49	Table 4.A.46
13	6	14	4	Table 4.A.49	Table 4.A.42
14	5	14	6	Table 4.A.49	Table 4.A.43
15	6	14	8	Table 4.A.49	Table 4.A.44
16	8	14	12	Table 4.A.49	Table 4.A.46
17	9	14	14	Table 4.A.49	Table 4.A.47
18	6	15	4	Table 4.A.49	Table 4.A.42
19	5	15	6	Table 4.A.49	Table 4.A.43
20	6	15	8	Table 4.A.49	Table 4.A.44
21	8	15	12	Table 4.A.49	Table 4.A.46
22	10	15	15	Table 4.A.49	Table 4.A.48
23	8	16	12	Table 4.A.49	Table 4.A.46
24	10	16	16	Table 4.A.49	Table 4.A.48
25	9	17	14	Table 4.A.49	Table 4.A.47
26	10	17	17	Table 4.A.49	Table 4.A.48
27	10	18	18	Table 4.A.49	Table 4.A.48
28	12	19	19	Table 4.A.49	Table 4.A.48
29	12	20	20	Table 4.A.49	Table 4.A.48
30	12	21	21	Table 4.A.49	Table 4.A.48
31	12	22	22	Table 4.A.49	Table 4.A.48

Table 4.A.30 – Scalefactor model parameters

scf_model	Largest Differential ArModel	Model listed in
0	0	not used
1	3	Table 4.A.35
2	7	Table 4.A.36
3	15	Table 4.A.37
4	15	Table 4.A.38
5	31	Table 4.A.39
6	31	Table 4.A.40
7	63	Table 4.A.41

Table 4.A.31 – BSAC cband_si parameters

cband_si	MSB plane	Table listed in	cband_si	MSB plane	Table listed in
0	0	-	12	6	Table 4.A.65
1	1	Table 4.A.54	13	7	Table 4.A.66
2	1	Table 4.A.55	14	7	Table 4.A.67
3	2	Table 4.A.56	15	8	Table 4.A.68
4	2	Table 4.A.57	16	9	Table 4.A.69
5	3	Table 4.A.58	17	10	Table 4.A.70
6	3	Table 4.A.59	18	11	Table 4.A.71
7	4	Table 4.A.60	19	12	Table 4.A.72
8	4	Table 4.A.61	20	13	Table 4.A.73
9	5	Table 4.A.62	21	14	Table 4.A.74
10	5	Table 4.A.63	22	15	Table 4.A.75
11	6	Table 4.A.64			

Table 4.A.32 – Position of probability value in probability table

				h	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				g	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				f	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				e	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
a	b	c	d																	
0	x	x	x		0	15	22	29	32	39	42	45								
1	x	x	0		1	16	23	30					46	53	56	59				
	x	x	1		2	17	24	31					46	53	56	59				
2	x	0	0		3	18			33	40			47	54			60	63		
	x	0	1		4	19			33	40			48	55			60	63		
	x	1	0		5	20			34	41			47	54			60	63		
	x	1	1		6	21			34	41			48	55			60	63		
3	0	0	0		7		25		35		43		49		57		61		64	
	0	0	1		8		25		36		43		50		57		62		64	
	0	1	0		9		26		35		43		51		58		61		64	
	0	1	1		10		26		36		43		52		58		62		64	
	1	0	0		11		27		37		44		49		57		61		64	
	1	0	1		12		27		38		44		50		57		62		64	
	1	1	0		13		28		37		44		51		58		61		64	
	1	1	1		14		28		38		44		52		58		62		64	

where, i = spectral index

a = i % 4

b = the sliced bit of (i-3)th spectral data whose significance is same with that of i-th spectral data

c = the sliced bit of (i-2)th spectral data whose significance is same with that of i-th spectral data

d = the sliced bit of (i-1)th spectral data whose significance is same with that of i-th spectral data

e = whether the higher bits of the (i-a+3)th spectral data whose sigificance is larger than that of i-th spectral data is nonzero (1) or zero(0)

f = whether the higher bits of the (i-a+2)th spectral data whose sigificance is larger than that of i-th spectral data is nonzero (1) or zero(0)

g = whether the higher bits of the (i-a+1)th spectral data whose sigificance is larger than that of i-th spectral data is nonzero (1) or zero(0)

h = whether the higher bits of the (i-a)th spectral data whose sigificance is larger than that of i-th spectral data is nonzero (1) or zero(0)

Table 4.A.33 – The minimum probability(min_p0) in proportion to the available length of the layer

Available length	1	2	3	4	5	6	7	8	9	10	11	12	13
min_p0 (hexadecimal)	2000	1000	800	400	200	100	80	40	20	10	8	4	2

Table 4.A.34 – The maximum probability(max_p0) in proportion to the available length of the layer

Available length	1	2	3	4	5	6	7	8	9	10	11	12	13
max_p0 (hexadecimal)	2000	3000	3800	3C00	3E00	3F00	3F80	3FC0	3FE0	3FF0	3FF8	3FFC	3FFE

Table 4.A.35 – Scalefactor arithmetic model 1

size	cumulative frequencies (hexadecimal)
4	752, 3cd, 14d, 0,

Table 4.A.36 – Scalefactor arithmetic model 2

size	cumulative frequencies (hexadecimal)
8	112f, de7, a8b, 7c1, 47a, 23a, d4, 0,

Table 4.A.37 – Scalefactor arithmetic model 3

size	cumulative frequencies (hexadecimal)
16	1f67, 1c5f, 18d8, 1555, 1215, eb4, adc, 742, 408, 1e6, df, 52, 32, 23, c, 0,

Table 4.A.38 – Scalefactor arithmetic model 4

size	cumulative frequencies (hexadecimal)
16	250f, 22b8, 2053, 1deb, 1b05, 186d, 15df, 12d9, f77, c01, 833, 50d, 245, 8c, 33, 0,

Table 4.A.39 – Scalefactor arithmetic model 5

size	cumulative frequencies (hexadecimal)
32	8a8, 74e, 639, 588, 48c, 3cf, 32e, 272, 1bc, 13e, e4, 97, 69, 43, 2f, 29, 20, 1b, 18, 15, 12, f, d, c, a, 9, 7, 6, 4, 3, 1, 0,

Table 4.A.40 – Scalefactor arithmetic model 6

size	cumulative frequencies (hexadecimal)
32	c2a, 99f, 809, 6ec, 603, 53d, 491, 40e, 394, 30a, 2a5, 259, 202, 1bc, 170, 133, 102, c9, 97, 73, 4f, 37, 22, 16, f, b, 9, 7, 5, 3, 1, 0,

Table 4.A.41 – Scalefactor arithmetic model 7

size	cumulative frequencies (hexadecimal)							
64	3b5e,	3a90,	39d3,	387c,	3702,	3566,	33a7,	321c,
	2f90,	2cf2,	29fe,	26fa,	23e4,	20df,	1e0d,	1ac4,
	1804,	159a,	131e,	10e7,	e5b,	c9c,	b78,	a21,
	8fd,	7b7,	6b5,	62c,	55d,	4f6,	4d4,	44b,
	38e,	2e2,	29d,	236,	225,	1f2,	1cf,	1ad,
	19c,	179,	168,	157,	146,	135,	123,	112,
	101,	f0,	df,	ce,	bc,	ab,	9a,	89,
	78,	67,	55,	44,	33,	22,	11,	0,

Table 4.A.42 – cband_si arithmetic model 0

size	cumulative frequencies (hexadecimal)							
5	3ef6,	3b59,	1b12,	12a3,	0,			

Table 4.A.43 – cband_si arithmetic model 1

size	cumulative frequencies (hexadecimal)							
7	3d51,	33ae,	1cff,	fb7,	7e4,	22b,	0,	

Table 4.A.44 – cband_si arithmetic model 2

size	cumulative frequencies (hexadecimal)							
9	3a47,	2aec,	1e05,	1336,	e7d,	860,	5e0,	44a,
	0,							

Table 4.A.45 – cband_si arithmetic model 3

size	cumulative frequencies (hexadecimal)							
11	36be,	27ae,	20f4,	1749,	14d5,	d46,	ad3,	888,
	519,	20b,	0,					

Table 4.A.46 – cband_si arithmetic model 4

size	cumulative frequencies (hexadecimal)							
13	3983,	2e77,	2b03,	1ee8,	1df9,	1307,	11e4,	b4d,
	94c,	497,	445,	40,	0,			

Table 4.A.47 – cband_si arithmetic model 5

size	cumulative frequencies (hexadecimal)							
15	306f,	249e,	1f56,	1843,	161a,	102d,	f6c,	c81,
	af2,	7a8,	71a,	454,	413,	16,	0,	

Table 4.A.48 – cband_si arithmetic model 6

size	cumulative frequencies (hexadecimal)							
23	31af,	2001,	162d,	127e,	f05,	c34,	b8f,	a61,
	955,	825,	7dd,	6a9,	688,	55b,	54b,	2f7,
	198,	77,	10,	c,	8,	4,	0,	

Table 4.A.49 – cband_si arithmetic model for 0th coding band

size	cumulative frequencies (hexadecimal)							
23	3ff8, 3074, 30,	3ff0, 2bcf, 28,	3fe8, 231b, 20,	3fe0, 13db, 18,	3fd7, d51, 10,	3f31, 603, 8,	3cd7, 44c, 0,	3bc9, 80,

Table 4.A.50 – MS_used model

size	Cumulative frequencies (hexadecimal)	
2	2CCD,	0,

Table 4.A.51 – stereo_info model

size	Cumulative frequencies(hexadecimal)			
4	3666,	1000,	666,	0,

Table 4.A.52 – noise_flag arithmetic model

size	Cumulative frequencies(hexadecimal)	
2	2000,	0,

Table 4.A.53 – noise_mode arithmetic model

size	cumulative frequencies(hexadecimal)			
4	3000,	2000,	1000,	0,

Table 4.A.54 – BSAC probability table 1

(MSB plane = 1)

Significance	Probability Value of symbol ‚0‘ (Hexadecimal)							
1	3900, 3000,	3a00, 3600,	2f00, 2d00,	3b00, 3900,	2f00, 2f00,	3700, 3700,	2c00, 2c00,	3b00,

Table 4.A.55 – BSAC probability table 2

(MSB plane = 1)

Significance	Probability Value of symbol ‚0‘ (Hexadecimal)							
1	2800, 2700,	2800, 2800,	2500, 2400,	2900, 2800,	2600, 2500,	2700, 2600,	2300, 2200,	2a00,

Table 4.A.56 – BSAC probability table 3

(MSB plane = 2)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)							
2	zero	3d00, 3200,	3d00, 3b00,	3300, 3100,	3d00, 3e00,	3300, 3700,	3b00, 3c00,	3300, 3300,	3d00,
1	zero	3700, 2500, 2c00, 1a00, 1e00, 1300, 2300, 1400, 2200,	3a00, 2b00, 1d00, 1d00, 1f00, 1a00, 3600, 2100,	2800, 2400, 2200, 1900, 1c00, 2000, 2800, 2200,	3b00, 3100, 1a00, 1c00, 2b00, 1800, 3100, 1000,	2600, 2300, 1c00, 1e00, 2400, 2300, 2500, 1e00,	2c00, 2900, 1600, 2c00, 2900, 2500, 1400, 3000,	2400, 2300, 2700, 2400, 2700, 1f00, 1200, 2600,	3a00, 3000, 2200, 1900, 2400, 2c00, 1800, 1200,
	non-zero	3100,							

Table 4.A.57 – BSAC probability table 4

(MSB plane = 2)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)							
2	zero	3900, 3000,	3a00, 3500,	2e00, 2c00,	3a00, 3600,	2f00, 2b00,	3400, 3100,	2a00, 2500,	3a00,
1	zero	1e00, 1e00, 1a00, 1700, 1700, 1c00, 1a00, 1400, 1400,	1d00, 1e00, 1800, 1700, 1800, 1500, 1e00, 1600,	1c00, 1a00, 1800, 1900, 1600, 1600, 1800, 1500,	1d00, 1e00, 1800, 1800, 1c00, 0f00, 1c00, 1700,	1c00, 1c00, 1700, 1600, 1700, 1800, 1b00, 1600,	1d00, 1d00, 1700, 1700, 1900, 1400, 1500, 1b00,	1b00, 1b00, 1800, 1600, 1700, 1700, 1300, 1800,	1d00, 1a00, 1a00, 1500, 1500, 1a00, 1500, 1400,
	non-zero	3600,							

Table 4.A.58 – BSAC probability table 5

(MSB plane = 3)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)							
3	zero	3d00, 3500,	3d00, 3c00,	3200, 3500,	3d00, 3f00,	3300, 3b00,	3d00, 3f00,	3600, 3d00,	3d00,
2	zero	3c00, 2b00, 3400, 1a00, 2600, 1800, 3000, 1900, 3100,	3d00, 3400, 2400, 2a00, 2500, 1600, 3c00, 2900,	2b00, 2b00, 2a00, 2200, 2700, 2900, 3300, 2a00,	3d00, 3800, 1c00, 2b00, 3500, 2500, 3b00, 2400,	2900, 2b00, 1f00, 2a00, 2d00, 3100, 3400, 2700,	3500, 3700, 1600, 3500, 3800, 2c00, 1700, 3c00,	2c00, 2a00, 3500, 2600, 3200, 2300, 1a00, 3600,	3d00, 3900, 2500, 1a00, 2e00, 3600, 1c00, 1d00,
	non-zero	3100,							
1	zero	3400, 2500, 2300, 1b00, 1900, 1200, 1e00, 1300, 1a00,	3800, 2d00, 1a00, 1c00, 1b00, 1400, 3000, 1e00,	2700, 2000, 1a00, 1b00, 1a00, 1a00, 2900, 1f00,	3900, 3300, 1b00, 1a00, 1d00, 1300, 2d00, 1100,	2700, 2000, 1800, 1800, 1e00, 1c00, 2500, 1900,	2f00, 2900, 1700, 1d00, 1f00, 1b00, 1300, 2100,	2200, 1e00, 1e00, 1b00, 1b00, 1900, 1700, 1e00,	3800, 2b00, 1c00, 1800, 1e00, 2000, 1400, 1500,
	non-zero	2a00,	2b00,	2800,					

Table 4.A.59 – BSAC probability table 6

(MSB plane = 3)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)							
3	zero	3800, 2d00,	3a00, 3600,	2d00, 2b00,	3a00, 3a00,	2d00, 2800,	3600, 3600,	2d00, 2700,	3a00,
2	zero	2b00, 2500, 2900, 1f00, 1d00, 1800, 2800, 1c00, 1a00,	3000, 2b00, 2300, 1d00, 1f00, 1a00, 2f00, 1e00,	2500, 2400, 2200, 2200, 1f00, 1d00, 2300, 2100,	2f00, 2d00, 1e00, 1b00, 2900, 2000, 2f00, 1700,	2600, 2500, 1b00, 1800, 2600, 1c00, 2600, 2200,	2d00, 2800, 1900, 2100, 2a00, 1a00, 1d00, 2300,	2400, 2500, 2600, 2100, 2100, 1e00, 1700, 2300,	3000, 2a00, 2300, 1d00, 2300, 2900, 1d00, 1400,
	non-zero	3000,							
1	zero	1900, 1900, 1700, 1600, 1300, 1400, 1600, 1300, 1300,	1900, 1600, 1500, 1600, 1600, 1400, 1f00, 1400,	1900, 1800, 1500, 1200, 1600, 1500, 1a00, 1300,	1b00, 1e00, 1500, 1300, 1c00, 1400, 1e00, 1100,	1700, 1900, 1700, 1200, 1400, 1300, 1800, 1500,	1b00, 1a00, 1400, 1600, 1700, 1300, 1700, 1600,	1a00, 1700, 1900, 1500, 1600, 1500, 1600, 1500,	1000, 1b00, 1700, 1500, 1400, 1800, 1600, 1200,
	non-zero	2b00,	2800,	2700,					

Table 4.A.60 – BSAC probability table 7

(MSB plane = 4)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)
MSB	zero	3d00, 3d00, 3500, 3e00, 3500, 3f00, 3b00, 3e00, 3200, 3f00, 3a00, 3f00, 3d00, 3f00, 3b00,
MSB-1	zero	3f00, 3f00, 3200, 3f00, 3500, 3e00, 3700, 3f00, 2d00, 3c00, 3000, 3f00, 3700, 3e00, 3400, 3f00, 3900, 2600, 2f00, 1e00, 2400, 1500, 3700, 3100, 1b00, 2600, 2300, 3a00, 3900, 3e00, 2b00, 2200, 2800, 2f00, 2500, 3e00, 3700, 3e00, 3d00, 3900, 1a00, 3300, 2500, 2800, 3c00, 3800, 2c00, 3d00, 3800, 3f00, 3b00, 3f00, 3a00, 1e00, 1b00, 1800, 1800, 3b00, 3a00, 1200, 2f00, 3f00, 3b00, 1b00, 3500,
	non-zero	2F00,
MSB-2	zero	3c00, 3e00, 3000, 3e00, 3100, 3a00, 3100, 3d00, 2c00, 3900, 2e00, 3c00, 2d00, 3c00, 3100, 3d00, 3100 2100, 2c00, 2600, 2800, 1d00, 2b00, 2800, 2800, 2400, 2200, 2100, 2300, 2d00, 2500, 1f00, 2100, 2b00, 2700, 3200, 2d00, 3400, 2a00, 3500, 1800, 1800, 1f00, 1e00, 2e00, 2a00, 2400, 3000, 2b00, 3e00, 3d00, 3d00, 3a00, 1e00, 2b00, 2600, 1900, 3400, 3500, 1c00, 2600, 3300, 2a00, 1c00, 2b00,
	non-zero	2800, 2900, 2400,
others	zero	3500, 3b00, 2900, 3b00, 2a00, 3100, 2700, 3b00, 2600, 2f00, 2400, 3400, 2300, 2d00, 2000, 3300, 2700, 1c00, 2400, 1c00, 1c00, 1900, 2700, 2800, 1b00, 1d00, 2000, 1b00, 1a00, 2300, 1d00, 1700, 1e00, 2400, 2100, 2b00, 2100, 2800, 2000, 2300, 1b00, 1500, 1b00, 1400, 1a00, 1a00, 2000, 2a00, 2200, 3700, 2f00, 3200, 2a00, 1700, 1700, 1600, 1900, 2500, 2300, 1500, 1900, 2500, 2200, 1400, 1b00,
	non-zero	2d00, 2500, 2300, 2500, 2500, 2600, 2400,

Table 4.A.61 – BSAC probability table 8

(MSB plane = 4)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)
MSB	zero	3b00, 3c00, 3400, 3c00, 3400, 3a00, 3000, 3c00, 3200, 3a00, 3100, 3c00, 3000, 3900, 2f00,
MSB-1	zero	3500, 3800, 2c00, 3900, 2c00, 3400, 2b00, 3800, 2e00, 3400, 2d00, 3600, 2a00, 3300, 2800, 3100, 3100, 2600, 2900, 2000, 2300, 1f00, 2d00, 2600, 2000, 2600, 2300, 2500, 2100, 2c00, 2400, 1d00, 2500, 2400, 2400, 3000, 2800, 3000, 2900, 2200, 1e00, 1c00, 2500, 1d00, 2300, 2300, 2500, 3300, 2c00, 3700, 2b00, 3400, 2c00, 1e00, 1c00, 2100, 1b00, 2900, 2a00, 1d00, 2600, 3200, 2a00, 2000, 2400,
	non-zero	3200,
		2900, 2e00, 2600, 2f00, 2600, 2d00, 2600, 2e00, 2500, 2b00, 2600, 2f00, 2300, 2a00, 2300, 2800,

MSB-2	zero	2800, 1c00, 1e00, 1a00, 2500, 1a00, 1c00,	2100, 2100, 2100, 1a00, 2d00, 1b00,	2400, 2200, 2100, 2100, 2700, 1d00,	2000, 1d00, 2900, 2100, 2a00, 1800,	2000, 1c00, 2200, 1c00, 2300, 2000,	1b00, 1f00, 2300, 1c00, 1c00, 2300,	2400, 1c00, 2100, 1f00, 1d00, 1f00,	1f00, 1900, 1c00, 2700, 1a00, 1900,
	non-zero	2b00,	2900,	2800,					
others	zero	1c00, 1f00, 1a00, 1900, 1600, 1500, 1a00, 1400, 1400,	1e00, 1f00, 1900, 1700, 1800, 1500, 2300, 1800,	1b00, 1900, 1800, 1800, 1a00, 1600, 1c00, 1500,	1e00, 2000, 1900, 1700, 1c00, 1600, 1d00, 1300,	1c00, 1a00, 1800, 1800, 1c00, 1500, 1a00, 1700,	1e00, 1f00, 1600, 1600, 1c00, 1400, 1600, 1900,	1900, 1700, 1900, 1700, 1700, 1700, 1600, 1600,	1a00, 1b00, 1a00, 1400, 1700, 1b00, 1500, 1400,
	non-zero	2800,	2500,	2500,	2700,	2500,	2600,	2500,	

Table 4.A.62 – BSAC probability table 9

(MSB plane = 5)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)							
MSB	zero	3d00, 3400,	3e00, 3e00,	3300, 3500,	3e00, 3f00,	3500, 3d00,	3e00, 3f00,	3700, 3c00,	3e00,
MSB-1	zero	same as BSAC probability Table 7 (see Table 4.A.60)							
	non-zero	2e00,							
MSB-2	zero	same as BSAC probability table 7 (seeTable 4.A.60)							
	non-zero	2900,	2a00,	2700,					
MSB-3	zero	same as BSAC probability table 7 (see Table 4.A.60)							
	non-zero	2d00,	2500,	2400,	2500,	2400,	2500,	2300,	
others	zero	same as BSAC probability table 7 (see Table 4.A.60)							
	non-zero	2800, 2200,	2500, 2200,	2300, 2200,	2300, 2100,	2200, 2000,	2200, 2200,	2200, 2100,	2200,

Table 4.A.63 – BSAC probability table 10

(MSB plane = 5)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)							
MSB	zero	3b00, 3400,	3c00, 3900,	3400, 2f00,	3c00, 3c00,	3200, 2d00,	3900, 3700,	2e00, 2d00,	3d00,
MSB-1	zero	same as BSAC probability table 8 (see Table 4.A.61)							
	non-zero	3100,							
MSB-2	zero	same as BSAC probability table 8 (see Table 4.A.61)							
	non-zero	2b00,	2a00,	2900,					
MSB-3	zero	same as BSAC probability table 8 (see Table 4.A.61)							
	non-zero	2700,	2600,	2500,	2500,	2500,	2200,	2200,	
others	zero	same as BSAC probability table 8 (see Table 4.A.61)							
	non-zero	2200, 2200,	2300, 2200,	2300, 2200,	2300, 2200,	2200, 2200,	2300, 2000,	2200, 2100,	2300, 2200

Table 4.A.64 – BSAC probability Table 11

Same as BSAC probability Table 9, but MSB plane = 6

Table 4.A.65 – BSAC probability Table 12

Same as BSAC probability Table 10, but MSB plane = 6

Table 4.A.66 – BSAC probability Table 13

Same as BSAC probability Table 9, but MSB plane = 7

Table 4.A.67 – BSAC probability Table 14

Same as BSAC probability Table 10, but MSB plane = 7

Table 4.A.68 – BSAC probability Table 15

Same as BSAC probability Table 9, but MSB plane = 8

Table 4.A.69 – BSAC probability Table 16

Same as BSAC probability Table 9, but MSB plane = 9

Table 4.A.70 – BSAC probability Table 17

Same as BSAC probability Table 9, but MSB plane = 10

Table 4.A.71 – BSAC probability Table 18

Same as BSAC probability Table 9, but MSB plane = 11

Table 4.A.72 – BSAC probability Table 19

Same as BSAC probability Table 9, but MSB plane = 12

Table 4.A.73 – BSAC probability Table 20

Same as BSAC probability Table 9, but MSB plane = 13

Table 4.A.74 – BSAC probability Table 21

Same as BSAC probability Table 9, but MSB plane = 14

Table 4.A.75 – BSAC probability Table 22

Same as BSAC probability Table 9, but MSB plane = 15

4.A.6 Tables for SBR

4.A.6.1 SBR Huffman tables

The function *sbr_huff_dec()* is used as:

data = sbr_huff_dec(t_huff, codeword),

where *t_huff* is the selected Huffman table and *codeword* is the word read from the bitstream payload. The returned value *data,* is a Huffman table index corresponding to a specific code word, with the largest absolute value (LAV) of the table subtracted.

Huffman table overview:

Table 4.A.76

table name	df_env_flag	df_noise_flag	amp_res	LAV	Notes
t_huffman_env_1_5dB	0	dc	0	60	Note 1
f_huffman_env_1_5dB	1	dc	0	60	
t_huffman_env_bal_1_5dB	0	dc	0	24	
f_huffman_env_bal_1_5dB	1	dc	0	24	
t_huffman_env_3_0dB	0	dc	1	31	
f_huffman_env_3_0dB	1	dc	1	31	
t_huffman_env_bal_3_0dB	0	dc	1	12	
f_huffman_env_bal_3_0dB	1	dc	1	12	
t_huffman_noise_3_0dB	dc	0	dc	31	
f_huffman_noise_3_0dB	dc	1	dc	31	Note 2
t_huffman_noise_bal_3_0dB	dc	0	dc	12	
f_huffman_noise_bal_3_0dB	dc	1	dc	12	Note 2
Note 1: dc (don't care), indicates that the variable is not relevant. Note 2: The Huffman tables of f_huffman_noise_3_0dB and f_huffman_noise_bal_3_0dB are the same as for f_huffman_env_3_0dB and f_huffman_env_bal_3_0dB, respectively.					

Table 4.A.77 – t_huffman_env_1_5dB

index	Length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x00000012	0x0003FFD6	61	0x00000003	0x00000004
1	0x00000012	0x0003FFD7	62	0x00000004	0x0000000C
2	0x00000012	0x0003FFD8	63	0x00000005	0x0000001C
3	0x00000012	0x0003FFD9	64	0x00000006	0x0000003C
4	0x00000012	0x0003FFDA	65	0x00000007	0x0000007C
5	0x00000012	0x0003FFDB	66	0x00000008	0x000000FC
6	0x00000013	0x0007FFB8	67	0x00000009	0x000001FC
7	0x00000013	0x0007FFB9	68	0x0000000A	0x000003FD
8	0x00000013	0x0007FFBA	69	0x0000000C	0x00000FFA
9	0x00000013	0x0007FFBB	70	0x0000000D	0x00001FF8
10	0x00000013	0x0007FFBC	71	0x0000000E	0x00003FF6
11	0x00000013	0x0007FFBD	72	0x0000000E	0x00003FF8
12	0x00000013	0x0007FFBE	73	0x0000000F	0x00007FF5
13	0x00000013	0x0007FFBF	74	0x00000010	0x0000FFEF
14	0x00000013	0x0007FFC0	75	0x00000011	0x0001FFE8
15	0x00000013	0x0007FFC1	76	0x00000010	0x0000FFF2
16	0x00000013	0x0007FFC2	77	0x00000013	0x0007FFD4
17	0x00000013	0x0007FFC3	78	0x00000013	0x0007FFD5
18	0x00000013	0x0007FFC4	79	0x00000013	0x0007FFD6
19	0x00000013	0x0007FFC5	80	0x00000013	0x0007FFD7
20	0x00000013	0x0007FFC6	81	0x00000013	0x0007FFD8
21	0x00000013	0x0007FFC7	82	0x00000013	0x0007FFD9
22	0x00000013	0x0007FFC8	83	0x00000013	0x0007FFDA
23	0x00000013	0x0007FFC9	84	0x00000013	0x0007FFDB
24	0x00000013	0x0007FFCA	85	0x00000013	0x0007FFDC
25	0x00000013	0x0007FFCB	86	0x00000013	0x0007FFDD
26	0x00000013	0x0007FFCC	87	0x00000013	0x0007FFDE
27	0x00000013	0x0007FFCD	88	0x00000013	0x0007FFDF
28	0x00000013	0x0007FFCE	89	0x00000013	0x0007FFE0
29	0x00000013	0x0007FFCF	90	0x00000013	0x0007FFE1
30	0x00000013	0x0007FFD0	91	0x00000013	0x0007FFE2
31	0x00000013	0x0007FFD1	92	0x00000013	0x0007FFE3
32	0x00000013	0x0007FFD2	93	0x00000013	0x0007FFE4
33	0x00000013	0x0007FFD3	94	0x00000013	0x0007FFE5
34	0x00000011	0x0001FFE6	95	0x00000013	0x0007FFE6
35	0x00000012	0x0003FFD4	96	0x00000013	0x0007FFE7
36	0x00000010	0x0000FFF0	97	0x00000013	0x0007FFE8
37	0x00000011	0x0001FFE9	98	0x00000013	0x0007FFE9
38	0x00000012	0x0003FFD5	99	0x00000013	0x0007FFEA
39	0x00000011	0x0001FFE7	100	0x00000013	0x0007FFEB
40	0x00000010	0x0000FFF1	101	0x00000013	0x0007FFEC
41	0x00000010	0x0000FFEC	102	0x00000013	0x0007FFED
42	0x00000010	0x0000FFED	103	0x00000013	0x0007FFEE
43	0x00000010	0x0000FFEE	104	0x00000013	0x0007FFEF
44	0x0000000F	0x00007FF4	105	0x00000013	0x0007FFF0
45	0x0000000E	0x00003FF9	106	0x00000013	0x0007FFF1
46	0x0000000E	0x00003FF7	107	0x00000013	0x0007FFF2
47	0x0000000D	0x00001FFA	108	0x00000013	0x0007FFF3
48	0x0000000D	0x00001FF9	109	0x00000013	0x0007FFF4

49	0x0000000C	0x00000FFB	110	0x00000013	0x0007FFF5
50	0x0000000B	0x000007FC	111	0x00000013	0x0007FFF6
51	0x0000000A	0x000003FC	112	0x00000013	0x0007FFF7
52	0x00000009	0x000001FD	113	0x00000013	0x0007FFF8
53	0x00000008	0x000000FD	114	0x00000013	0x0007FFF9
54	0x00000007	0x0000007D	115	0x00000013	0x0007FFFA
55	0x00000006	0x0000003D	116	0x00000013	0x0007FFFB
56	0x00000005	0x0000001D	117	0x00000013	0x0007FFFC
57	0x00000004	0x0000000D	118	0x00000013	0x0007FFFD
58	0x00000003	0x00000005	119	0x00000013	0x0007FFFE
59	0x00000002	0x00000001	120	0x00000013	0x0007FFFF
60	0x00000002	0x00000000			

Table 4.A.78 – f_huffman_env_1_5dB

index	length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x00000013	0x0007FFE7	61	0x00000003	0x00000004
1	0x00000013	0x0007FFE8	62	0x00000004	0x0000000D
2	0x00000014	0x000FFFD2	63	0x00000005	0x0000001D
3	0x00000014	0x000FFFD3	64	0x00000006	0x0000003D
4	0x00000014	0x000FFFD4	65	0x00000008	0x000000FA
5	0x00000014	0x000FFFD5	66	0x00000008	0x000000FC
6	0x00000014	0x000FFFD6	67	0x00000009	0x000001FB
7	0x00000014	0x000FFFD7	68	0x0000000A	0x000003FA
8	0x00000014	0x000FFFD8	69	0x0000000B	0x000007F8
9	0x00000013	0x0007FFDA	70	0x0000000B	0x000007FA
10	0x00000014	0x000FFFD9	71	0x0000000B	0x000007FB
11	0x00000014	0x000FFFDA	72	0x0000000C	0x00000FF9
12	0x00000014	0x000FFFDB	73	0x0000000C	0x00000FFB
13	0x00000014	0x000FFFDC	74	0x0000000D	0x00001FF8
14	0x00000013	0x0007FFDB	75	0x0000000D	0x00001FFB
15	0x00000014	0x000FFFDD	76	0x0000000E	0x00003FF8
16	0x00000013	0x0007FFDC	77	0x0000000E	0x00003FF9
17	0x00000013	0x0007FFDD	78	0x00000010	0x0000FFF1
18	0x00000014	0x000FFFDE	79	0x00000010	0x0000FFF2
19	0x00000012	0x0003FFE4	80	0x00000011	0x0001FFEA
20	0x00000014	0x000FFFDF	81	0x00000011	0x0001FFEB
21	0x00000014	0x000FFFE0	82	0x00000012	0x0003FFE1
22	0x00000014	0x000FFFE1	83	0x00000012	0x0003FFE2
23	0x00000013	0x0007FFDE	84	0x00000012	0x0003FFEA
24	0x00000014	0x000FFFE2	85	0x00000012	0x0003FFE3
25	0x00000014	0x000FFFE3	86	0x00000012	0x0003FFE6
26	0x00000014	0x000FFFE4	87	0x00000012	0x0003FFE7
27	0x00000013	0x0007FFDF	88	0x00000012	0x0003FFEB
28	0x00000014	0x000FFFE5	89	0x00000014	0x000FFFE6
29	0x00000013	0x0007FFE0	90	0x00000013	0x0007FFE2
30	0x00000012	0x0003FFE8	91	0x00000014	0x000FFFE7
31	0x00000013	0x0007FFE1	92	0x00000014	0x000FFFE8

32	0x00000012	0x0003FFE0	93	0x00000014	0x000FFFE9
33	0x00000012	0x0003FFE9	94	0x00000014	0x000FFFEA
34	0x00000011	0x0001FFEF	95	0x00000014	0x000FFFEB
35	0x00000012	0x0003FFE5	96	0x00000014	0x000FFFEC
36	0x00000011	0x0001FFEC	97	0x00000013	0x0007FFE3
37	0x00000011	0x0001FFED	98	0x00000014	0x000FFFED
38	0x00000011	0x0001FFEE	99	0x00000014	0x000FFFEE
39	0x00000010	0x0000FFF4	100	0x00000014	0x000FFFEF
40	0x00000010	0x0000FFF3	101	0x00000014	0x000FFFF0
41	0x00000010	0x0000FFF0	102	0x00000013	0x0007FFE4
42	0x0000000F	0x00007FF7	103	0x00000014	0x000FFFF1
43	0x0000000F	0x00007FF6	104	0x00000012	0x0003FFEC
44	0x0000000E	0x00003FFA	105	0x00000014	0x000FFFF2
45	0x0000000D	0x00001FFA	106	0x00000014	0x000FFFF3
46	0x0000000D	0x00001FF9	107	0x00000013	0x0007FFE5
47	0x0000000C	0x00000FFA	108	0x00000013	0x0007FFE6
48	0x0000000C	0x00000FF8	109	0x00000014	0x000FFFF4
49	0x0000000B	0x000007F9	110	0x00000014	0x000FFFF5
50	0x0000000A	0x000003FB	111	0x00000014	0x000FFFF6
51	0x00000009	0x000001FC	112	0x00000014	0x000FFFF7
52	0x00000009	0x000001FA	113	0x00000014	0x000FFFF8
53	0x00000008	0x000000FB	114	0x00000014	0x000FFFF9
54	0x00000007	0x0000007C	115	0x00000014	0x000FFFFA
55	0x00000006	0x0000003C	116	0x00000014	0x000FFFFB
56	0x00000005	0x0000001C	117	0x00000014	0x000FFFFC
57	0x00000004	0x0000000C	118	0x00000014	0x000FFFFD
58	0x00000003	0x00000005	119	0x00000014	0x000FFFFE
59	0x00000002	0x00000001	120	0x00000014	0x000FFFFF
60	0x00000002	0x00000000			

Table 4.A.79 – t_huffman_env_bal_1_5dB

index	length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x00000010	0x0000FFE4	25	0x00000002	0x00000002
1	0x00000010	0x0000FFE5	26	0x00000004	0x0000000E
2	0x00000010	0x0000FFE6	27	0x00000006	0x0000003E
3	0x00000010	0x0000FFE7	28	0x00000008	0x000000FE
4	0x00000010	0x0000FFE8	29	0x0000000B	0x000007FD
5	0x00000010	0x0000FFE9	30	0x0000000C	0x00000FFD
6	0x00000010	0x0000FFEA	31	0x0000000F	0x00007FF0
7	0x00000010	0x0000FFEB	32	0x00000010	0x0000FFE3
8	0x00000010	0x0000FFEC	33	0x00000010	0x0000FFF5
9	0x00000010	0x0000FFED	34	0x00000010	0x0000FFF6
10	0x00000010	0x0000FFEE	35	0x00000010	0x0000FFF7
11	0x00000010	0x0000FFEF	36	0x00000010	0x0000FFF8
12	0x00000010	0x0000FFF0	37	0x00000010	0x0000FFF9
13	0x00000010	0x0000FFF1	38	0x00000010	0x0000FFFA
14	0x00000010	0x0000FFF2	39	0x00000011	0x0001FFF6
15	0x00000010	0x0000FFF3	40	0x00000011	0x0001FFF7
16	0x00000010	0x0000FFF4	41	0x00000011	0x0001FFF8
17	0x00000010	0x0000FFE2	42	0x00000011	0x0001FFF9
18	0x0000000C	0x00000FFC	43	0x00000011	0x0001FFFA
19	0x0000000B	0x000007FC	44	0x00000011	0x0001FFFB
20	0x00000009	0x000001FE	45	0x00000011	0x0001FFFC
21	0x00000007	0x0000007E	46	0x00000011	0x0001FFFD
22	0x00000005	0x0000001E	47	0x00000011	0x0001FFFE
23	0x00000003	0x00000006	48	0x00000011	0x0001FFFF
24	0x00000001	0x00000000			

Table 4.A.80 – f_huffman_env_bal_1_5dB

index	length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x00000012	0x0003FFE2	25	0x00000003	0x00000006
1	0x00000012	0x0003FFE3	26	0x00000005	0x0000001E
2	0x00000012	0x0003FFE4	27	0x00000006	0x0000003E
3	0x00000012	0x0003FFE5	28	0x00000009	0x000001FE
4	0x00000012	0x0003FFE6	29	0x0000000B	0x000007FD
5	0x00000012	0x0003FFE7	30	0x0000000C	0x00000FFE
6	0x00000012	0x0003FFE8	31	0x0000000F	0x00007FFA
7	0x00000012	0x0003FFE9	32	0x00000010	0x0000FFF6
8	0x00000012	0x0003FFEA	33	0x00000012	0x0003FFF1
9	0x00000012	0x0003FFEB	34	0x00000012	0x0003FFF2
10	0x00000012	0x0003FFEC	35	0x00000012	0x0003FFF3
11	0x00000012	0x0003FFED	36	0x00000012	0x0003FFF4
12	0x00000012	0x0003FFEE	37	0x00000012	0x0003FFF5
13	0x00000012	0x0003FFEF	38	0x00000012	0x0003FFF6
14	0x00000012	0x0003FFF0	39	0x00000012	0x0003FFF7
15	0x00000010	0x0000FFF7	40	0x00000012	0x0003FFF8
16	0x00000011	0x0001FFF0	41	0x00000012	0x0003FFF9
17	0x0000000E	0x00003FFC	42	0x00000012	0x0003FFFA
18	0x0000000B	0x000007FE	43	0x00000012	0x0003FFFB
19	0x0000000B	0x000007FC	44	0x00000012	0x0003FFFC
20	0x00000008	0x000000FE	45	0x00000012	0x0003FFFD
21	0x00000007	0x0000007E	46	0x00000012	0x0003FFFE
22	0x00000004	0x0000000E	47	0x00000013	0x0007FFFE
23	0x00000002	0x00000002	48	0x00000013	0x0007FFFF
24	0x00000001	0x00000000			

Table 4.A.81 – t_huffman_env_3_0dB

index	length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x00000012	0x0003FFED	32	0x00000003	0x00000006
1	0x00000012	0x0003FFEE	33	0x00000005	0x0000001E
2	0x00000013	0x0007FFDE	34	0x00000007	0x0000007E
3	0x00000013	0x0007FFDF	35	0x00000009	0x000001FE
4	0x00000013	0x0007FFE0	36	0x0000000B	0x000007FD
5	0x00000013	0x0007FFE1	37	0x0000000D	0x00001FFB
6	0x00000013	0x0007FFE2	38	0x0000000E	0x00003FF9
7	0x00000013	0x0007FFE3	39	0x0000000E	0x00003FFC
8	0x00000013	0x0007FFE4	40	0x0000000F	0x00007FFA
9	0x00000013	0x0007FFE5	41	0x00000010	0x0000FFF6
10	0x00000013	0x0007FFE6	42	0x00000011	0x0001FFF5
11	0x00000013	0x0007FFE7	43	0x00000012	0x0003FFEC
12	0x00000013	0x0007FFE8	44	0x00000013	0x0007FFED
13	0x00000013	0x0007FFE9	45	0x00000013	0x0007FFEE
14	0x00000013	0x0007FFEA	46	0x00000013	0x0007FFEF
15	0x00000013	0x0007FFEB	47	0x00000013	0x0007FFF0
16	0x00000013	0x0007FFEC	48	0x00000013	0x0007FFF1
17	0x00000011	0x0001FFF4	49	0x00000013	0x0007FFF2
18	0x00000010	0x0000FFF7	50	0x00000013	0x0007FFF3
19	0x00000010	0x0000FFF9	51	0x00000013	0x0007FFF4
20	0x00000010	0x0000FFF8	52	0x00000013	0x0007FFF5
21	0x0000000E	0x00003FFB	53	0x00000013	0x0007FFF6
22	0x0000000E	0x00003FFA	54	0x00000013	0x0007FFF7
23	0x0000000E	0x00003FF8	55	0x00000013	0x0007FFF8
24	0x0000000D	0x00001FFA	56	0x00000013	0x0007FFF9
25	0x0000000C	0x00000FFC	57	0x00000013	0x0007FFFA
26	0x0000000B	0x000007FC	58	0x00000013	0x0007FFFB
27	0x00000008	0x000000FE	59	0x00000013	0x0007FFFC
28	0x00000006	0x0000003E	60	0x00000013	0x0007FFFD
29	0x00000004	0x0000000E	61	0x00000013	0x0007FFFE
30	0x00000002	0x00000002	62	0x00000013	0x0007FFFF
31	0x00000001	0x00000000			

Table 4.A.82 – f_huffman_env_3_0dB

index	length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x00000014	0x000FFFF0	32	0x00000003	0x00000006
1	0x00000014	0x000FFFF1	33	0x00000005	0x0000001E
2	0x00000014	0x000FFFF2	34	0x00000008	0x000000FC
3	0x00000014	0x000FFFF3	35	0x00000009	0x000001FC
4	0x00000014	0x000FFFF4	36	0x0000000A	0x000003FC
5	0x00000014	0x000FFFF5	37	0x0000000B	0x000007FC
6	0x00000014	0x000FFFF6	38	0x0000000C	0x00000FFC
7	0x00000012	0x0003FFF3	39	0x0000000D	0x00001FFC
8	0x00000013	0x0007FFF5	40	0x0000000E	0x00003FFA
9	0x00000013	0x0007FFEE	41	0x0000000F	0x00007FF9
10	0x00000013	0x0007FFEF	42	0x0000000F	0x00007FFA
11	0x00000013	0x0007FFF6	43	0x00000010	0x0000FFF8
12	0x00000012	0x0003FFF4	44	0x00000010	0x0000FFF9
13	0x00000012	0x0003FFF2	45	0x00000011	0x0001FFF6
14	0x00000014	0x000FFFF7	46	0x00000011	0x0001FFF7
15	0x00000013	0x0007FFF0	47	0x00000012	0x0003FFF5
16	0x00000011	0x0001FFF5	48	0x00000012	0x0003FFF6
17	0x00000012	0x0003FFF0	49	0x00000012	0x0003FFF1
18	0x00000011	0x0001FFF4	50	0x00000014	0x000FFFF8
19	0x00000010	0x0000FFF7	51	0x00000013	0x0007FFF1
20	0x00000010	0x0000FFF6	52	0x00000013	0x0007FFF2
21	0x0000000F	0x00007FF8	53	0x00000013	0x0007FFF3
22	0x0000000E	0x00003FFB	54	0x00000014	0x000FFFF9
23	0x0000000C	0x00000FFD	55	0x00000013	0x0007FFF7
24	0x0000000B	0x000007FD	56	0x00000013	0x0007FFF4
25	0x0000000A	0x000003FD	57	0x00000014	0x000FFFFA
26	0x00000009	0x000001FD	58	0x00000014	0x000FFFFB
27	0x00000008	0x000000FD	59	0x00000014	0x000FFFFC
28	0x00000006	0x0000003E	60	0x00000014	0x000FFFFD
29	0x00000004	0x0000000E	61	0x00000014	0x000FFFFE
30	0x00000002	0x00000002	62	0x00000014	0x000FFFFF
31	0x00000001	0x00000000			

Table 4.A.83 – t_huffman_env_bal_3_0dB

index	length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x0000000D	0x00001FF2	13	0x00000002	0x00000002
1	0x0000000D	0x00001FF3	14	0x00000005	0x0000001E
2	0x0000000D	0x00001FF4	15	0x00000006	0x0000003E
3	0x0000000D	0x00001FF5	16	0x00000009	0x000001FE
4	0x0000000D	0x00001FF6	17	0x0000000D	0x00001FF9
5	0x0000000D	0x00001FF7	18	0x0000000D	0x00001FFA
6	0x0000000D	0x00001FF8	19	0x0000000D	0x00001FFB
7	0x0000000C	0x00000FF8	20	0x0000000D	0x00001FFC
8	0x00000008	0x000000FE	21	0x0000000D	0x00001FFD
9	0x00000007	0x0000007E	22	0x0000000D	0x00001FFE
10	0x00000004	0x0000000E	23	0x0000000E	0x00003FFE
11	0x00000003	0x00000006	24	0x0000000E	0x00003FFF
12	0x00000001	0x00000000			

Table 4.A.84 – f_huffman_env_bal_3_0dB

index	length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x0000000D	0x00001FF7	13	0x00000003	0x00000006
1	0x0000000D	0x00001FF8	14	0x00000005	0x0000001E
2	0x0000000D	0x00001FF9	15	0x00000006	0x0000003E
3	0x0000000D	0x00001FFA	16	0x00000009	0x000001FE
4	0x0000000D	0x00001FFB	17	0x0000000C	0x00000FFA
5	0x0000000E	0x00003FF8	18	0x0000000D	0x00001FF6
6	0x0000000E	0x00003FF9	19	0x0000000E	0x00003FFA
7	0x0000000B	0x000007FC	20	0x0000000E	0x00003FFB
8	0x00000008	0x000000FE	21	0x0000000E	0x00003FFC
9	0x00000007	0x0000007E	22	0x0000000E	0x00003FFD
10	0x00000004	0x0000000E	23	0x0000000E	0x00003FFE
11	0x00000002	0x00000002	24	0x0000000E	0x00003FFF
12	0x00000001	0x00000000			

Table 4.A.85 – t_huffman_noise_3_0dB

index	length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x0000000D	0x00001FCE	32	0x00000002	0x00000002
1	0x0000000D	0x00001FCF	33	0x00000005	0x0000001E
2	0x0000000D	0x00001FD0	34	0x00000008	0x000000FC
3	0x0000000D	0x00001FD1	35	0x0000000A	0x000003F8
4	0x0000000D	0x00001FD2	36	0x0000000D	0x00001FCC
5	0x0000000D	0x00001FD3	37	0x0000000D	0x00001FE8
6	0x0000000D	0x00001FD4	38	0x0000000D	0x00001FE9
7	0x0000000D	0x00001FD5	39	0x0000000D	0x00001FEA
8	0x0000000D	0x00001FD6	40	0x0000000D	0x00001FEB
9	0x0000000D	0x00001FD7	41	0x0000000D	0x00001FEC
10	0x0000000D	0x00001FD8	42	0x0000000D	0x00001FCD
11	0x0000000D	0x00001FD9	43	0x0000000D	0x00001FED
12	0x0000000D	0x00001FDA	44	0x0000000D	0x00001FEE
13	0x0000000D	0x00001FDB	45	0x0000000D	0x00001FEF
14	0x0000000D	0x00001FDC	46	0x0000000D	0x00001FF0
15	0x0000000D	0x00001FDD	47	0x0000000D	0x00001FF1
16	0x0000000D	0x00001FDE	48	0x0000000D	0x00001FF2
17	0x0000000D	0x00001FDF	49	0x0000000D	0x00001FF3
18	0x0000000D	0x00001FE0	50	0x0000000D	0x00001FF4
19	0x0000000D	0x00001FE1	51	0x0000000D	0x00001FF5
20	0x0000000D	0x00001FE2	52	0x0000000D	0x00001FF6
21	0x0000000D	0x00001FE3	53	0x0000000D	0x00001FF7
22	0x0000000D	0x00001FE4	54	0x0000000D	0x00001FF8
23	0x0000000D	0x00001FE5	55	0x0000000D	0x00001FF9
24	0x0000000D	0x00001FE6	56	0x0000000D	0x00001FFA
25	0x0000000D	0x00001FE7	57	0x0000000D	0x00001FFB
26	0x0000000B	0x000007F2	58	0x0000000D	0x00001FFC
27	0x00000008	0x000000FD	59	0x0000000D	0x00001FFD
28	0x00000006	0x0000003E	60	0x0000000D	0x00001FFE
29	0x00000004	0x0000000E	61	0x0000000E	0x00003FFE
30	0x00000003	0x00000006	62	0x0000000E	0x00003FFF
31	0x00000001	0x00000000			

Table 4.A.86 – t_huffman_noise_bal_3_0dB

index	length (hexadecimal)	codeword (hexadecimal)	index	length (hexadecimal)	codeword (hexadecimal)
0	0x00000008	0x000000EC	13	0x00000003	0x00000006
1	0x00000008	0x000000ED	14	0x00000006	0x0000003A
2	0x00000008	0x000000EE	15	0x00000008	0x000000F6
3	0x00000008	0x000000EF	16	0x00000008	0x000000F7
4	0x00000008	0x000000F0	17	0x00000008	0x000000F8
5	0x00000008	0x000000F1	18	0x00000008	0x000000F9
6	0x00000008	0x000000F2	19	0x00000008	0x000000FA
7	0x00000008	0x000000F3	20	0x00000008	0x000000FB
8	0x00000008	0x000000F4	21	0x00000008	0x000000FC
9	0x00000008	0x000000F5	22	0x00000008	0x000000FD
10	0x00000005	0x0000001C	23	0x00000008	0x000000FE
11	0x00000002	0x00000002	24	0x00000008	0x000000FF
12	0x00000001	0x00000000			

4.A.6.2 Miscellaneous SBR tables

Table 4.A.87 – Coefficients c[i] of the QMF bank window

i	c[i]	i	c[i]	i	c[i]
0	0.0000000000	214	0.0019765601	428	0.0117623832
1	-0.0005525286	215	-0.0032086896	429	0.0163701258
2	-0.0005617692	216	-0.0085711749	430	0.0207997072
3	-0.0004947518	217	-0.0141288827	431	0.0250307561
4	-0.0004875227	218	-0.0198834129	432	0.0290824006
5	-0.0004893791	219	-0.0258227288	433	0.0329583930
6	-0.0005040714	220	-0.0319531274	434	0.0366418116
7	-0.0005226564	221	-0.0382776572	435	0.0401458278
8	-0.0005466565	222	-0.0447806821	436	0.0434768782
9	-0.0005677802	223	-0.0514804176	437	0.0466303305
10	-0.0005870930	224	-0.0583705326	438	0.0495978676
11	-0.0006132747	225	-0.0654409853	439	0.0524093821
12	-0.0006312493	226	-0.0726943300	440	0.0550460034
13	-0.0006540333	227	-0.0801372934	441	0.0575152691
14	-0.0006777690	228	-0.0877547536	442	0.0598166570
15	-0.0006941614	229	-0.0955533352	443	0.0619602779
16	-0.0007157736	230	-0.1035329531	444	0.0639444805
17	-0.0007255043	231	-0.1116826931	445	0.0657690668
18	-0.0007440941	232	-0.1200077984	446	0.0674525021
19	-0.0007490598	233	-0.1285002850	447	0.0689664013
20	-0.0007681371	234	-0.1371551761	448	0.0703533073
21	-0.0007724848	235	-0.1459766491	449	0.0715826364
22	-0.0007834332	236	-0.1549607071	450	0.0726774642
23	-0.0007779869	237	-0.1640958855	451	0.0736406005
24	-0.0007803664	238	-0.1733808172	452	0.0744664394
25	-0.0007801449	239	-0.1828172548	453	0.0751576255
26	-0.0007757977	240	-0.1923966745	454	0.0757305756
27	-0.0007630793	241	-0.2021250176	455	0.0761748321
28	-0.0007530001	242	-0.2119735853	456	0.0765050718
29	-0.0007319357	243	-0.2219652696	457	0.0767204924
30	-0.0007215391	244	-0.2320690870	458	0.0768230011
31	-0.0006917937	245	-0.2423016884	459	0.0768173975
32	-0.0006650415	246	-0.2526480309	460	0.0767093490
33	-0.0006341594	247	-0.2631053299	461	0.0764992170
34	-0.0005946118	248	-0.2736634040	462	0.0761992479
35	-0.0005564576	249	-0.2843214189	463	0.0758008358
36	-0.0005145572	250	-0.2950716717	464	0.0753137336
37	-0.0004606325	251	-0.3059098575	465	0.0747452558
38	-0.0004095121	252	-0.3168278913	466	0.0741003642
39	-0.0003501175	253	-0.3278113727	467	0.0733620255
40	-0.0002896981	254	-0.3388722693	468	0.0725682583
41	-0.0002098337	255	-0.3499914122	469	0.0717002673
42	-0.0001446380	256	0.3611589903	470	0.0707628710
43	-0.0000617334	257	0.3723795546	471	0.0697630244
44	0.0000134949	258	0.3836350013	472	0.0687043828
45	0.0001094383	259	0.3949211761	473	0.0676075985
46	0.0002043017	260	0.4062317676	474	0.0664367512

47	0.0002949531	261	0.4175696896	475	0.0652247106
48	0.0004026540	262	0.4289119920	476	0.0639715898
49	0.0005107388	263	0.4402553754	477	0.0626857808
50	0.0006239376	264	0.4515996535	478	0.0613455171
51	0.0007458025	265	0.4629308085	479	0.0599837480
52	0.0008608443	266	0.4742453214	480	0.0585915683
53	0.0009885988	267	0.4855253091	481	0.0571616450
54	0.0011250155	268	0.4967708254	482	0.0557173648
55	0.0012577884	269	0.5079817500	483	0.0542452768
56	0.0013902494	270	0.5191234970	484	0.0527630746
57	0.0015443219	271	0.5302240895	485	0.0512556155
58	0.0016868083	272	0.5412553448	486	0.0497385755
59	0.0018348265	273	0.5522051258	487	0.0482165720
60	0.0019841140	274	0.5630789140	488	0.0466843027
61	0.0021461583	275	0.5738524131	489	0.0451488405
62	0.0023017254	276	0.5845403235	490	0.0436097542
63	0.0024625616	277	0.5951123086	491	0.0420649094
64	0.0026201758	278	0.6055783538	492	0.0405349170
65	0.0027870464	279	0.6159109932	493	0.0390053679
66	0.0029469447	280	0.6261242695	494	0.0374812850
67	0.0031125420	281	0.6361980107	495	0.0359697560
68	0.0032739613	282	0.6461269695	496	0.0344620948
69	0.0034418874	283	0.6559016302	497	0.0329754081
70	0.0036008268	284	0.6655139880	498	0.0315017608
71	0.0037603922	285	0.6749663190	499	0.0300502657
72	0.0039207432	286	0.6842353293	500	0.0286072173
73	0.0040819753	287	0.6933282376	501	0.0271859429
74	0.0042264269	288	0.7022388719	502	0.0257875847
75	0.0043730719	289	0.7109410426	503	0.0244160992
76	0.0045209852	290	0.7194462634	504	0.0230680169
77	0.0046606460	291	0.7277448900	505	0.0217467550
78	0.0047932560	292	0.7358211758	506	0.0204531793
79	0.0049137603	293	0.7436827863	507	0.0191872431
80	0.0050393022	294	0.7513137456	508	0.0179433381
81	0.0051407353	295	0.7587080760	509	0.0167324712
82	0.0052461166	296	0.7658674865	510	0.0155405553
83	0.0053471681	297	0.7727780881	511	0.0143904666
84	0.0054196775	298	0.7794287519	512	-0.0132718220
85	0.0054876040	299	0.7858353120	513	-0.0121849995
86	0.0055475714	300	0.7919735841	514	-0.0111315548
87	0.0055938023	301	0.7978466413	515	-0.0101150215
88	0.0056220643	302	0.8034485751	516	-0.0091325329
89	0.0056455196	303	0.8087695004	517	-0.0081798233
90	0.0056389199	304	0.8138191270	518	-0.0072615816
91	0.0056266114	305	0.8185776004	519	-0.0063792293
92	0.0055917128	306	0.8230419890	520	-0.0055337211
93	0.0055404363	307	0.8272275347	521	-0.0047222596
94	0.0054753783	308	0.8311038457	522	-0.0039401124
95	0.0053838975	309	0.8346937361	523	-0.0031933778
96	0.0052715758	310	0.8379717337	524	-0.0024826723
97	0.0051382275	311	0.8409541392	525	-0.0018039472
98	0.0049839687	312	0.8436238281	526	-0.0011568135

99	0.0048109469	313	0.8459818469	527	-0.0005464280
100	0.0046039530	314	0.8480315777	528	0.0000276045
101	0.0043801861	315	0.8497805198	529	0.0005832264
102	0.0041251642	316	0.8511971524	530	0.0010902329
103	0.0038456408	317	0.8523047035	531	0.0015784682
104	0.0035401246	318	0.8531020949	532	0.0020274176
105	0.0032091885	319	0.8535720573	533	0.0024508540
106	0.0028446757	320	0.8537385600	534	0.0028446757
107	0.0024508540	321	0.8535720573	535	0.0032091885
108	0.0020274176	322	0.8531020949	536	0.0035401246
109	0.0015784682	323	0.8523047035	537	0.0038456408
110	0.0010902329	324	0.8511971524	538	0.0041251642
111	0.0005832264	325	0.8497805198	539	0.0043801861
112	0.0000276045	326	0.8480315777	540	0.0046039530
113	-0.0005464280	327	0.8459818469	541	0.0048109469
114	-0.0011568135	328	0.8436238281	542	0.0049839687
115	-0.0018039472	329	0.8409541392	543	0.0051382275
116	-0.0024826723	330	0.8379717337	544	0.0052715758
117	-0.0031933778	331	0.8346937361	545	0.0053838975
118	-0.0039401124	332	0.8311038457	546	0.0054753783
119	-0.0047222596	333	0.8272275347	547	0.0055404363
120	-0.0055337211	334	0.8230419890	548	0.0055917128
121	-0.0063792293	335	0.8185776004	549	0.0056266114
122	-0.0072615816	336	0.8138191270	550	0.0056389199
123	-0.0081798233	337	0.8087695004	551	0.0056455196
124	-0.0091325329	338	0.8034485751	552	0.0056220643
125	-0.0101150215	339	0.7978466413	553	0.0055938023
126	-0.0111315548	340	0.7919735841	554	0.0055475714
127	-0.0121849995	341	0.7858353120	555	0.0054876040
128	0.0132718220	342	0.7794287519	556	0.0054196775
129	0.0143904666	343	0.7727780881	557	0.0053471681
130	0.0155405553	344	0.7658674865	558	0.0052461166
131	0.0167324712	345	0.7587080760	559	0.0051407353
132	0.0179433381	346	0.7513137456	560	0.0050393022
133	0.0191872431	347	0.7436827863	561	0.0049137603
134	0.0204531793	348	0.7358211758	562	0.0047932560
135	0.0217467550	349	0.7277448900	563	0.0046606460
136	0.0230680169	350	0.7194462634	564	0.0045209852
137	0.0244160992	351	0.7109410426	565	0.0043730719
138	0.0257875847	352	0.7022388719	566	0.0042264269
139	0.0271859429	353	0.6933282376	567	0.0040819753
140	0.0286072173	354	0.6842353293	568	0.0039207432
141	0.0300502657	355	0.6749663190	569	0.0037603922
142	0.0315017608	356	0.6655139880	570	0.0036008268
143	0.0329754081	357	0.6559016302	571	0.0034418874
144	0.0344620948	358	0.6461269695	572	0.0032739613
145	0.0359697560	359	0.6361980107	573	0.0031125420
146	0.0374812850	360	0.6261242695	574	0.0029469447
147	0.0390053679	361	0.6159109932	575	0.0027870464
148	0.0405349170	362	0.6055783538	576	0.0026201758
149	0.0420649094	363	0.5951123086	577	0.0024625616
150	0.0436097542	364	0.5845403235	578	0.0023017254

151	0.0451488405	365	0.5738524131	579	0.0021461583
152	0.0466843027	366	0.5630789140	580	0.0019841140
153	0.0482165720	367	0.5522051258	581	0.0018348265
154	0.0497385755	368	0.5412553448	582	0.0016868083
155	0.0512556155	369	0.5302240895	583	0.0015443219
156	0.0527630746	370	0.5191234970	584	0.0013902494
157	0.0542452768	371	0.5079817500	585	0.0012577884
158	0.0557173648	372	0.4967708254	586	0.0011250155
159	0.0571616450	373	0.4855253091	587	0.0009885988
160	0.0585915683	374	0.4742453214	588	0.0008608443
161	0.0599837480	375	0.4629308085	589	0.0007458025
162	0.0613455171	376	0.4515996535	590	0.0006239376
163	0.0626857808	377	0.4402553754	591	0.0005107388
164	0.0639715898	378	0.4289119920	592	0.0004026540
165	0.0652247106	379	0.4175696896	593	0.0002949531
166	0.0664367512	380	0.4062317676	594	0.0002043017
167	0.0676075985	381	0.3949211761	595	0.0001094383
168	0.0687043828	382	0.3836350013	596	0.0000134949
169	0.0697630244	383	0.3723795546	597	-0.0000617334
170	0.0707628710	384	-0.3611589903	598	-0.0001446380
171	0.0717002673	385	-0.3499914122	599	-0.0002098337
172	0.0725682583	386	-0.3388722693	600	-0.0002896981
173	0.0733620255	387	-0.3278113727	601	-0.0003501175
174	0.0741003642	388	-0.3168278913	602	-0.0004095121
175	0.0747452558	389	-0.3059098575	603	-0.0004606325
176	0.0753137336	390	-0.2950716717	604	-0.0005145572
177	0.0758008358	391	-0.2843214189	605	-0.0005564576
178	0.0761992479	392	-0.2736634040	606	-0.0005946118
179	0.0764992170	393	-0.2631053299	607	-0.0006341594
180	0.0767093490	394	-0.2526480309	608	-0.0006650415
181	0.0768173975	395	-0.2423016884	609	-0.0006917937
182	0.0768230011	396	-0.2320690870	610	-0.0007215391
183	0.0767204924	397	-0.2219652696	611	-0.0007319357
184	0.0765050718	398	-0.2119735853	612	-0.0007530001
185	0.0761748321	399	-0.2021250176	613	-0.0007630793
186	0.0757305756	400	-0.1923966745	614	-0.0007757977
187	0.0751576255	401	-0.1828172548	615	-0.0007801449
188	0.0744664394	402	-0.1733808172	616	-0.0007803664
189	0.0736406005	403	-0.1640958855	617	-0.0007779869
190	0.0726774642	404	-0.1549607071	618	-0.0007834332
191	0.0715826364	405	-0.1459766491	619	-0.0007724848
192	0.0703533073	406	-0.1371551761	620	-0.0007681371
193	0.0689664013	407	-0.1285002850	621	-0.0007490598
194	0.0674525021	408	-0.1200077984	622	-0.0007440941
195	0.0657690668	409	-0.1116826931	623	-0.0007255043
196	0.0639444805	410	-0.1035329531	624	-0.0007157736
197	0.0619602779	411	-0.0955533352	625	-0.0006941614
198	0.0598166570	412	-0.0877547536	626	-0.0006777690
199	0.0575152691	413	-0.0801372934	627	-0.0006540333
200	0.0550460034	414	-0.0726943300	628	-0.0006312493
201	0.0524093821	415	-0.0654409853	629	-0.0006132747
202	0.0495978676	416	-0.0583705326	630	-0.0005870930

203	0.0466303305	417	-0.0514804176	631	-0.0005677802
204	0.0434768782	418	-0.0447806821	632	-0.0005466565
205	0.0401458278	419	-0.0382776572	633	-0.0005226564
206	0.0366418116	420	-0.0319531274	634	-0.0005040714
207	0.0329583930	421	-0.0258227288	635	-0.0004893791
208	0.0290824006	422	-0.0198834129	636	-0.0004875227
209	0.0250307561	423	-0.0141288827	637	-0.0004947518
210	0.0207997072	424	-0.0085711749	638	-0.0005617692
211	0.0163701258	425	-0.0032086896	639	-0.000552528
212	0.0117623832	426	0.0019765601		
213	0.0069636862	427	0.0069636862		

Table 4.A.88 – Noise table $\mathbf{V}\left[\varphi_{Re,noise}(i), \varphi_{Im,noise}(i)\right]$

i	$\varphi_{re,noise}(i)$	$\varphi_{im,noise}(i)$	i	$\varphi_{re,noise}(i)$	$\varphi_{im,noise}(i)$
0	-0.99948153278296	-0.59483417516607	256	0.99570534804836	0.45844586038111
1	0.97113454393991	-0.67528515225647	257	-0.63431466947340	0.21079116459234
2	0.14130051758487	-0.95090983575689	258	-0.07706847005931	-0.89581437101329
3	-0.47005496701697	-0.37340549728647	259	0.98590090577724	0.88241721133981
4	0.80705063769351	0.29653668284408	260	0.80099335254678	-0.36851896710853
5	-0.38981478896926	0.89572605717087	261	0.78368131392666	0.45506999802597
6	-0.01053049862020	-0.66959058036166	262	0.08707806671691	0.80938994918745
7	-0.91266367957293	-0.11522938140034	263	-0.86811883080712	0.39347308654705
8	0.54840422910309	0.75221367176302	264	-0.39466529740375	-0.66809432114456
9	0.40009252867955	-0.98929400334421	265	0.97875325649683	-0.72467840967746
10	-0.99867974711855	-0.88147068645358	266	-0.95038560288864	0.89563219587625
11	-0.95531076805040	0.90908757154593	267	0.17005239424212	0.54683053962658
12	-0.45725933317144	-0.56716323646760	268	-0.76910792026848	-0.96226617549298
13	-0.72929675029275	-0.98008272727324	269	0.99743281016846	0.42697157037567
14	0.75622801399036	0.20950329995549	270	0.95437383549973	0.97002324109952
15	0.07069442601050	-0.78247898470706	271	0.99578905365569	-0.54106826257356
16	0.74496252926055	-0.91169004445807	272	0.28058259829990	-0.85361420634036
17	-0.96440182703856	-0.94739918296622	273	0.85256524470573	-0.64567607735589
18	0.30424629369539	-0.49438267012479	274	-0.50608540105128	-0.65846015480300
19	0.66565033746925	0.64652935542491	275	-0.97210735183243	-0.23095213067791
20	0.91697008020594	0.17514097332009	276	0.95424048234441	-0.99240147091219
21	-0.70774918760427	0.52548653416543	277	-0.96926570524023	0.73775654896574
22	-0.70051415345560	-0.45340028808763	278	0.30872163214726	0.41514960556126
23	-0.99496513054797	-0.90071908066973	279	-0.24523839572639	0.63206633394807
24	0.98164490790123	-0.77463155528697	280	-0.33813265086024	-0.38661779441897
25	-0.54671580548181	-0.02570928536004	281	-0.05826828420146	-0.06940774188029
26	-0.01689629065389	0.00287506445732	282	-0.22898461455054	0.97054853316316
27	-0.86110349531986	0.42548583726477	283	-0.18509915019881	0.47565762892084
28	-0.98892980586032	-0.87881132267556	284	-0.10488238045009	-0.87769947402394
29	0.51756627678691	0.66926784710139	285	-0.71886586182037	0.78030982480538
30	-0.99635026409640	-0.58107730574765	286	0.99793873738654	0.90041310491497
31	-0.99969370862163	0.98369989360250	287	0.57563307626120	-0.91034337352097
32	0.55266258627194	0.59449057465591	288	0.28909646383717	0.96307783970534
33	0.34581177741673	0.94879421061866	289	0.42188998312520	0.48148651230437
34	0.62664209577999	-0.74402970906471	290	0.93335049681047	-0.43537023883588
35	-0.77149701404973	-0.33883658042801	291	-0.97087374418267	0.86636445711364
36	-0.91592244254432	0.03687901376713	292	0.36722871286923	0.65291654172961
37	-0.76285492357887	-0.91371867919124	293	-0.81093025665696	0.08778370229363
38	0.79788337195331	-0.93180971199849	294	-0.26240603062237	-0.92774095379098
39	0.54473080610200	-0.11919206037186	295	0.83996497984604	0.55839849139647
40	-0.85639281671058	0.42429854760451	296	-0.99909615720225	-0.96024605713970
41	-0.92882402971423	0.27871809078609	297	0.74649464155061	0.12144893606462
42	-0.11708371046774	-0.99800843444966	298	-0.74774595569805	-0.26898062008959
43	0.21356749817493	-0.90716295627033	299	0.95781667469567	-0.79047927052628
44	-0.76191692573909	0.99768118356265	300	0.95472308713099	-0.08588776019550
45	0.98111043100884	-0.95854459734407	301	0.48708332746299	0.99999041579432
46	-0.85913269895572	0.95766566168880	302	0.46332038247497	0.10964126185063
47	-0.93307242253692	0.49431757696466	303	-0.76497004940162	0.89210929242238

48	0.30485754879632	-0.70540034357529	304	0.57397389364339	0.35289703373760
49	0.85289650925190	0.46766131791044	305	0.75374316974495	0.96705214651335
50	0.91328082618125	-0.99839597361769	306	-0.59174397685714	-0.89405370422752
51	-0.05890199924154	0.70741827819497	307	0.75087906691890	-0.29612672982396
52	0.28398686150148	0.34633555702188	308	-0.98607857336230	0.25034911730023
53	0.95258164539612	-0.54893416026939	309	-0.40761056640505	-0.90045573444695
54	-0.78566324168507	-0.75568541079691	310	0.66929266740477	0.98629493401748
55	-0.95789495447877	-0.20423194696966	311	-0.97463695257310	-0.00190223301301
56	0.82411158711197	0.96654618432562	312	0.90145509409859	0.99781390365446
57	-0.65185446735885	-0.88734990773289	313	-0.87259289048043	0.99233587353666
58	-0.93643603134666	0.99870790442385	314	-0.91529461447692	-0.15698707534206
59	0.91427159529618	-0.98290505544444	315	-0.03305738840705	-0.37205262859764
60	-0.70395684036886	0.58796798221039	316	0.07223051368337	-0.88805001733626
61	0.00563771969365	0.61768196727244	317	0.99498012188353	0.97094358113387
62	0.89065051931895	0.52783352697585	318	-0.74904939500519	0.99985483641521
63	-0.68683707712762	0.80806944710339	319	0.04585228574211	0.99812337444082
64	0.72165342518718	-0.69259857349564	320	-0.89054954257993	-0.31791913188064
65	-0.62928247730667	0.13627037407335	321	-0.83782144651251	0.97637632547466
66	0.29938434065514	-0.46051329682246	322	0.33454804933804	-0.86231516800408
67	-0.91781958879280	-0.74012716684186	323	-0.99707579362824	0.93237990079441
68	0.99298717043688	0.40816610075661	324	-0.22827527843994	0.18874759397997
69	0.82368298622748	-0.74036047190173	325	0.67248046289143	-0.03646211390569
70	-0.98512833386833	-0.99972330709594	326	-0.05146538187944	-0.92599700120679
71	-0.95915368242257	-0.99237800466040	327	0.99947295749905	0.93625229707912
72	-0.21411126572790	-0.93424819052545	328	0.66951124390363	0.98905825623893
73	-0.68821476106884	-0.26892306315457	329	-0.99602956559179	-0.44654715757688
74	0.91851997982317	0.09358228901785	330	0.82104905483590	0.99540741724928
75	-0.96062769559127	0.36099095133739	331	0.99186510988782	0.72023001312947
76	0.51646184922287	-0.71373332873917	332	-0.65284592392918	0.52186723253637
77	0.61130721139669	0.46950141175917	333	0.93885443798188	-0.74895312615259
78	0.47336129371299	-0.27333178296162	334	0.96735248738388	0.90891816978629
79	0.90998308703519	0.96715662938132	335	-0.22225968841114	0.57124029781228
80	0.44844799194357	0.99211574628306	336	-0.44132783753414	-0.92688840659280
81	0.66614891079092	0.96590176169121	337	-0.85694974219574	0.88844532719844
82	0.74922239129237	-0.89879858826087	338	0.91783042091762	-0.46356892383970
83	-0.99571588506485	0.52785521494349	339	0.72556974415690	-0.99899555770747
84	0.97401082477563	-0.16855870075190	340	-0.99711581834508	0.58211560180426
85	0.72683747733879	-0.48060774432251	341	0.77638976371966	0.94321834873819
86	0.95432193457128	0.68849603408441	342	0.07717324253925	0.58638399856595
87	-0.72962208425191	-0.76608443420917	343	-0.56049829194163	0.82522301569036
88	-0.85359479233537	0.88738125901579	344	0.98398893639988	0.39467440420569
89	-0.81412430338535	-0.97480768049637	345	0.47546946844938	0.68613044836811
90	-0.87930772356786	0.74748307690436	346	0.65675089314631	0.18331637134880
91	-0.71573331064977	-0.98570608178923	347	0.03273375457980	-0.74933109564108
92	0.83524300028228	0.83702537075163	348	-0.38684144784738	0.51337349030406
93	-0.48086065601423	-0.98848504923531	349	-0.97346267944545	-0.96549364384098
94	0.97139128574778	0.80093621198236	350	-0.53282156061942	-0.91423265091354
95	0.51992825347895	0.80247631400510	351	0.99817310731176	0.61133572482148
96	-0.00848591195325	-0.76670128000486	352	-0.50254500772635	-0.88829338134294
97	-0.70294374303036	0.55359910445577	353	0.01995873238855	0.85223515096765
98	-0.95894428168140	-0.43265504344783	354	0.99930381973804	0.94578896296649
99	0.97079252950321	0.09325857238682	355	0.82907767600783	-0.06323442598128

100	-0.92404293670797	0.85507704027855	356	-0.58660709669728	0.96840773806582
101	-0.69506469500450	0.98633412625459	357	-0.17573736667267	-0.48166920859485
102	0.26559203620024	0.73314307966524	358	0.83434292401346	-0.13023450646997
103	0.28038443336943	0.14537913654427	359	0.05946491307025	0.20511047074866
104	-0.74138124825523	0.99310339807762	360	0.81505484574602	-0.94685947861369
105	-0.01752795995444	-0.82616635284178	361	-0.44976380954860	0.40894572671545
106	-0.55126773094930	-0.98898543862153	362	-0.89746474625671	0.99846578838537
107	0.97960898850996	-0.94021446752851	363	0.39677256130792	-0.74854668609359
108	-0.99196309146936	0.67019017358456	364	-0.07588948563079	0.74096214084170
109	-0.67684928085260	0.12631491649378	365	0.76343198951445	0.41746629422634
110	0.09140039465500	-0.20537731453108	366	-0.74490104699626	0.94725911744610
111	-0.71658965751996	-0.97788200391224	367	0.64880119792759	0.41336660830571
112	0.81014640078925	0.53722648362443	368	0.62319537462542	-0.93098313552599
113	0.40616991671205	-0.26469008598449	369	0.42215817594807	-0.07712787385208
114	-0.67680188682972	0.94502052337695	370	0.02704554141885	-0.05417518053666
115	0.86849774348749	-0.18333598647899	371	0.80001773566818	0.91542195141039
116	-0.99500381284851	-0.02634122068550	372	-0.79351832348816	-0.36208897989136
117	0.84329189340667	0.10406957462213	373	0.63872359151636	0.08128252493444
118	-0.09215968531446	0.69540012101253	374	0.52890520960295	0.60048872455592
119	0.99956173327206	-0.12358542001404	375	0.74238552914587	0.04491915291044
120	-0.79732779473535	-0.91582524736159	376	0.99096131449250	-0.19451182854402
121	0.96349973642406	0.96640458041000	377	-0.80412329643109	-0.88513818199457
122	-0.79942778496547	0.64323902822857	378	-0.64612616129736	0.72198674804544
123	-0.11566039853896	0.28587846253726	379	0.11657770663191	-0.83662833815041
124	-0.39922954514662	0.94129601616966	380	-0.95053182488101	-0.96939905138082
125	0.99089197565987	-0.92062625581587	381	-0.62228872928622	0.82767262846661
126	0.28631285179909	-0.91035047143603	382	0.03004475787316	-0.99738896333384
127	-0.83302725605608	-0.67330410892084	383	-0.97987214341034	0.36526129686425
128	0.95404443402072	0.49162765398743	384	-0.99986980746200	-0.36021610299715
129	-0.06449863579434	0.03250560813135	385	0.89110648599879	-0.97894250343044
130	-0.99575054486311	0.42389784469507	386	0.10407960510582	0.77357793811619
131	-0.65501142790847	0.82546114655624	387	0.95964737821728	-0.35435818285502
132	-0.81254441908887	-0.51627234660629	388	0.50843233159162	0.96107691266205
133	-0.99646369485481	0.84490533520752	389	0.17006334670615	-0.76854025314829
134	0.00287840603348	0.64768261158166	390	0.25872675063360	0.99893303933816
135	0.70176989408455	-0.20453028573322	391	-0.01115998681937	0.98496019742444
136	0.96361882270190	0.40706967140989	392	-0.79598702973261	0.97138411318894
137	-0.68883758192426	0.91338958840772	393	-0.99264708948101	-0.99542822402536
138	-0.34875585502238	0.71472290693300	394	-0.99829663752818	0.01877138824311
139	0.91980081243087	0.66507455644919	395	-0.70801016548184	0.33680685948117
140	-0.99009048343881	0.85868021604848	396	-0.70467057786826	0.93272777501857
141	0.68865791458395	0.55660316809678	397	0.99846021905254	-0.98725746254433
142	-0.99484402129368	-0.20052559254934	398	-0.63364968534650	-0.16473594423746
143	0.94214511408023	-0.99696425367461	399	-0.16258217500792	-0.95939125400802
144	-0.67414626793544	0.49548221180078	400	-0.43645594360633	-0.94805030113284
145	-0.47339353684664	-0.85904328834047	401	-0.99848471702976	0.96245166923809
146	0.14323651387360	-0.94145598222488	402	-0.16796458968998	-0.98987511890470
147	-0.29268293575672	0.05759224927952	403	-0.87979225745213	-0.71725725041680
148	0.43793861458754	-0.78904969892724	404	0.44183099021786	-0.93568974498761
149	-0.36345126374441	0.64874435357162	405	0.93310180125532	-0.99913308068246
150	-0.08750604656825	0.97686944362527	406	-0.93941931782002	-0.56409379640356
151	-0.96495267812511	-0.53960305946511	407	-0.88590003188677	0.47624600491382

152	0.55526940659947	0.78891523734774	408	0.99971463703691	-0.83889954253462
153	0.73538215752630	0.96452072373404	409	-0.75376385639978	0.00814643438625
154	-0.30889773919437	-0.80664389776860	410	0.93887685615875	-0.11284528204636
155	0.03574995626194	-0.97325616900959	411	0.85126435782309	0.52349251543547
156	0.98720684660488	0.48409133691962	412	0.39701421446381	0.81779634174316
157	-0.81689296271203	-0.90827703628298	413	-0.37024464187437	-0.87071656222959
158	0.67866860118215	0.81284503870856	414	-0.36024828242896	0.34655735648287
159	-0.15808569732583	0.85279555024382	415	-0.93388812549209	-0.84476541096429
160	0.80723395114371	-0.24717418514605	416	-0.65298804552119	-0.18439575450921
161	0.47788757329038	-0.46333147839295	417	0.11960319006843	0.99899346780168
162	0.96367554763201	0.38486749303242	418	0.94292565553160	0.83163906518293
163	-0.99143875716818	-0.24945277239809	419	0.75081145286948	-0.35533223142265
164	0.83081876925833	-0.94780851414763	420	0.56721979748394	-0.24076836414499
165	-0.58753191905341	0.01290772389163	421	0.46857766746029	-0.30140233457198
166	0.95538108220960	-0.85557052096538	422	0.97312313923635	-0.99548191630031
167	-0.96490920476211	-0.64020970923102	423	-0.38299976567017	0.98516909715427
168	-0.97327101028521	0.12378128133110	424	0.41025800019463	0.02116736935734
169	0.91400366022124	0.57972471346930	425	0.09638062008048	0.04411984381457
170	-0.99925837363824	0.71084847864067	426	-0.85283249275397	0.91475563922421
171	-0.86875903507313	-0.20291699203564	427	0.88866808958124	-0.99735267083226
172	-0.26240034795124	-0.68264554369108	428	-0.48202429536989	-0.96805608884164
173	-0.24664412953388	-0.87642273115183	429	0.27572582416567	0.58634753335832
174	0.02416275806869	0.27192914288905	430	-0.65889129659168	0.58835634138583
175	0.82068619590515	-0.85087787994476	431	0.98838086953732	0.99994349600236
176	0.88547373760759	-0.89636802901469	432	-0.20651349620689	0.54593044066355
177	-0.18173078152226	-0.26152145156800	433	-0.62126416356920	-0.59893681700392
178	0.09355476558534	0.54845123045604	434	0.20320105410437	-0.86879180355289
179	-0.54668414224090	0.95980774020221	435	-0.97790548600584	0.96290806999242
180	0.37050990604091	-0.59910140383171	436	0.11112534735126	0.21484763313301
181	-0.70373594262891	0.91227665827081	437	-0.41368337314182	0.28216837680365
182	-0.34600785879594	-0.99441426144200	438	0.24133038992960	0.51294362630238
183	-0.68774481731008	-0.30238837956299	439	-0.66393410674885	-0.08249679629081
184	-0.26843291251234	0.83115668004362	440	-0.53697829178752	-0.97649903936228
185	0.49072334613242	-0.45359708737775	441	-0.97224737889348	0.22081333579837
186	0.38975993093975	0.95515358099121	442	0.87392477144549	-0.12796173740361
187	-0.97757125224150	0.05305894580606	443	0.19050361015753	0.01602615387195
188	-0.17325552859616	-0.92770672250494	444	-0.46353441212724	-0.95249041539006
189	0.99948035025744	0.58285545563426	445	-0.07064096339021	-0.94479803205886
190	-0.64946246527458	0.68645507104960	446	-0.92444085484466	-0.10457590187436
191	-0.12016920576437	-0.57147322153312	447	-0.83822593578728	-0.01695043208885
192	-0.58947456517751	-0.34847132454388	448	0.75214681811150	-0.99955681042665
193	-0.41815140454465	0.16276422358861	449	-0.42102998829339	0.99720941999394
194	0.99885650204884	0.11136095490444	450	-0.72094786237696	-0.35008961934255
195	-0.56649614128386	-0.90494866361587	451	0.78843311019251	0.52851398958271
196	0.94138021032330	0.35281916733018	452	0.97394027897442	-0.26695944086561
197	-0.75725076534641	0.53650549640587	453	0.99206463477946	-0.57010120849429
198	0.20541973692630	-0.94435144369918	454	0.76789609461795	-0.76519356730966
199	0.99980371023351	0.79835913565599	455	-0.82002421836409	-0.73530179553767
200	0.29078277605775	0.35393777921520	456	0.81924990025724	0.99698425250579
201	-0.62858772103030	0.38765693387102	457	-0.26719850873357	0.68903369776193
202	0.43440904467688	-0.98546330463232	458	-0.43311260380975	0.85321815947490
203	-0.98298583762390	0.21021524625209	459	0.99194979673836	0.91876249766422

204	0.19513029146934	-0.94239832251867	460	-0.80692001248487	-0.32627540663214
205	-0.95476662400101	0.98364554179143	461	0.43080003649976	-0.21919095636638
206	0.93379635304810	-0.70881994583682	462	0.67709491937357	-0.95478075822906
207	-0.85235410573336	-0.08342347966410	463	0.56151770568316	-0.70693811747778
208	-0.86425093011245	-0.45795025029466	464	0.10831862810749	-0.08628837174592
209	0.38879779059045	0.97274429344593	465	0.91229417540436	-0.65987351408410
210	0.92045124735495	-0.62433652524220	466	-0.48972893932274	0.56289246362686
211	0.89162532251878	0.54950955570563	467	-0.89033658689697	-0.71656563987082
212	-0.36834336949252	0.96458298020975	468	0.65269447475094	0.65916004833932
213	0.93891760988045	-0.89968353740388	469	0.67439478141121	-0.81684380846796
214	0.99267657565094	-0.03757034316958	470	-0.47770832416973	-0.16789556203025
215	-0.94063471614176	0.41332338538963	471	-0.99715979260878	-0.93565784007648
216	0.99740224117019	-0.16830494996370	472	-0.90889593602546	0.62034397054380
217	-0.35899413170555	-0.46633226649613	473	-0.06618622548177	-0.23812217221359
218	0.05237237274947	-0.25640361602661	474	0.99430266919728	0.18812555317553
219	0.36703583957424	-0.38653265641875	475	0.97686402381843	-0.28664534366620
220	0.91653180367913	-0.30587628726597	476	0.94813650221268	-0.97506640027128
221	0.69000803499316	0.90952171386132	477	-0.95434497492853	-0.79607978501983
222	-0.38658751133527	0.99501571208985	478	-0.49104783137150	0.32895214359663
223	-0.29250814029851	0.37444994344615	479	0.99881175120751	0.88993983831354
224	-0.60182204677608	0.86779651036123	480	0.50449166760303	-0.85995072408434
225	-0.97418588163217	0.96468523666475	481	0.47162891065108	-0.18680204049569
226	0.88461574003963	0.57508405276414	482	-0.62081581361840	0.75000676218956
227	0.05198933055162	0.21269661669964	483	-0.43867015250812	0.99998069244322
228	-0.53499621979720	0.97241553731237	484	0.98630563232075	-0.53578899600662
229	-0.49429560226497	0.98183865291903	485	-0.61510362277374	-0.89515019899997
230	-0.98935142339139	-0.40249159006933	486	-0.03841517601843	-0.69888815681179
231	-0.98081380091130	-0.72856895534041	487	-0.30102157304644	-0.07667808922205
232	-0.27338148835532	0.99950922447209	488	0.41881284182683	0.02188098922282
233	0.06310802338302	-0.54539587529618	489	-0.86135454941237	0.98947480909359
234	-0.20461677199539	-0.14209977628489	490	0.67226861393788	-0.13494389011014
235	0.66223843141647	0.72528579940326	491	-0.70737398842068	-0.76547349325992
236	-0.84764345483665	0.02372316801261	492	0.94044946687963	0.09026201157416
237	-0.89039863483811	0.88866581484602	493	-0.82386352534327	0.08924768823676
238	0.95903308477986	0.76744927173873	494	-0.32070666698656	0.50143421908753
239	0.73504123909879	-0.03747203173192	495	0.57593163224487	-0.98966422921509
240	-0.31744434966056	-0.36834111883652	496	-0.36326018419965	0.07440243123228
241	-0.34110827591623	0.40211222807691	497	0.99979044674350	-0.14130287347405
242	0.47803883714199	-0.39423219786288	498	-0.92366023326932	-0.97979298068180
243	0.98299195879514	0.01989791390047	499	-0.44607178518598	-0.54233252016394
244	-0.30963073129751	-0.18076720599336	500	0.44226800932956	0.71326756742752
245	0.99992588229018	-0.26281872094289	501	0.03671907158312	0.63606389366675
246	-0.93149731080767	-0.98313162570490	502	0.52175424682195	-0.85396826735705
247	0.99923472302773	-0.80142993767554	503	-0.94701139690956	-0.01826348194255
248	-0.26024169633417	-0.75999759855752	504	-0.98759606946049	0.82288714303073
249	-0.35712514743563	0.19298963768574	505	0.87434794743625	0.89399495655433
250	-0.99899084509530	0.74645156992493	506	-0.93412041758744	0.41374052024363
251	0.86557171579452	0.55593866696299	507	0.96063943315511	0.93116709541280
252	0.33408042438752	0.86185953874709	508	0.97534253457837	0.86150930812689
253	0.99010736374716	0.04602397576623	509	0.99642466504163	0.70190043427512
254	-0.66694269691195	-0.91643611810148	510	-0.94705089665984	-0.29580042814306
255	0.64016792079480	0.15649530836856	511	0.91599807087376	-0.98147830385781

Annex 4.B
(informative)

Encoder tools

4.B.1 Psychoacoustic model

See ISO/IEC13818-7 (13818-7:2005, subclause C.1 "Psychoacoustic Model").

4.B.2 Gain control

See ISO/IEC13818-7 (13818-7:2005, subclause C.2 "Gain Control ").

4.B.3 Filterbank and block switching

See ISO/IEC13818-7 (13818-7:2005, subclause C.3 "Filterbank and Block Switching").

4.B.4 Frequency domain prediction

See ISO/IEC13818-7 (13818-7:2005, subclause C.4 "Prediction").

4.B.5 Temporal noise shaping (TNS)

See ISO/IEC13818-7 (13818-7:2005, subclause C.5 "Temporal Noise Shaping (TNS)").

4.B.6 Joint coding

See ISO/IEC13818-7 (13818-7:2005, subclause C.6 "Joint Coding").

4.B.7 Quantization

See ISO/IEC13818-7 (13818-7:2005, subclause C.7 "Quantization").

4.B.8 Noiseless coding

See ISO/IEC13818-7 (13818-7:2005, subclause C.8 "Noiseless Coding").

4.B.9 Features of AAC dynamic range control

See ISO/IEC13818-7 (13818-7:2005, subclause C.9 " Features of AAC dynamic range control").

4.B.10 Long term prediction

The general structure of the perceptual audio coder with LTP is shown in Figure 4.B.1. First, the optimal LTP is estimated. Then, the predicted signals based on quantized data are obtained and corresponding error signals are calculated. Finally, the error signals are quantized and transmitted with the side information, such as predictor control and predictor parameters.

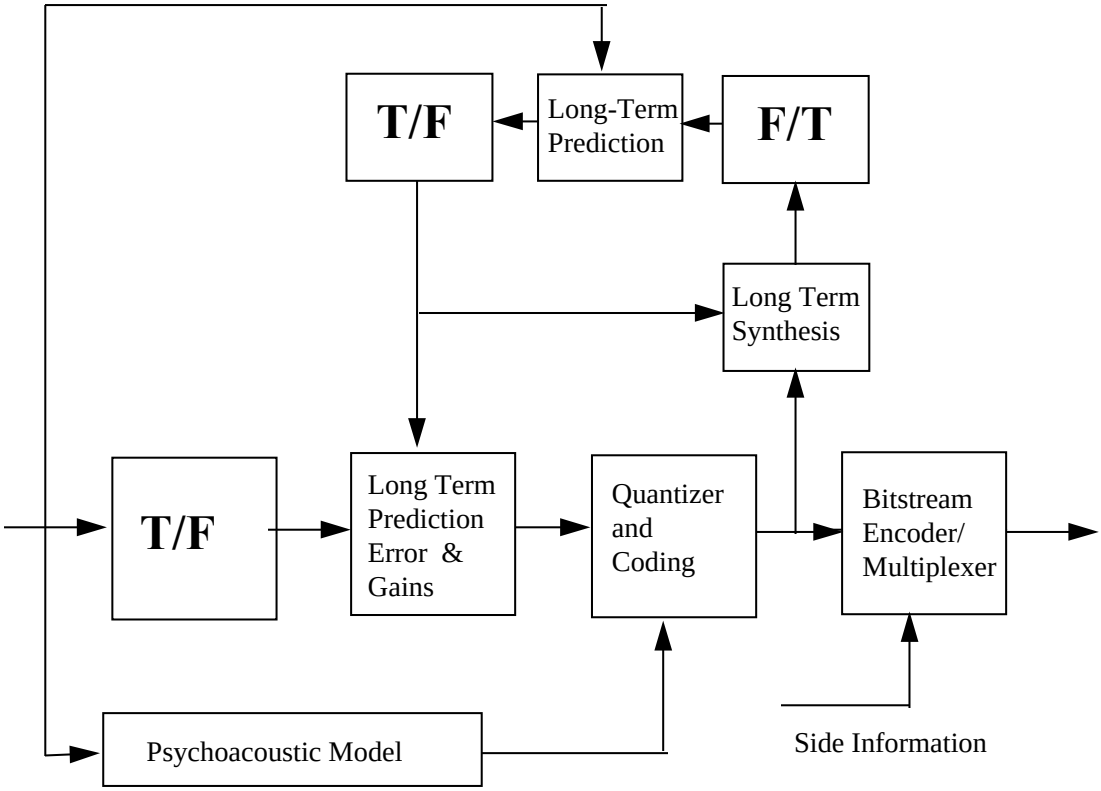


Figure 4.B.1 – Block diagram of the encoding process of Long Term Prediction

As the perceptual audio coder processes the signal frame by frame, it is obvious that when we want to quantize $\mathbf{X}_{m+1}$, the quantized data $\tilde{\mathbf{X}}_k, k = m, m-1, m-2, \ldots,$ are available both in encoder and decoder. In time domain, it means that the quantised time domain samples $\tilde{x}(i), i = mN, mN-1, \ldots,$ are available.

To remove the redundancy of the signal in present frame $m+1$ based on the previously quantized data, an LTP is used

$$P(z) = \sum_{k=-m_1}^{m_2} b_k z^{-(\alpha+k)}$$

where α represents a long delay in the range 1 to 2048. For $m_1 = m_2 = 0$, we have a one-tap predictor.

The parameters α and b_k are determined by minimizing the mean squared error over the whole window. The delay α is quantized with 11 bits with 2048 possible values in the range 1 to 2048. For prediction coefficients b_k are non-uniformly quantized to 3 bits. Due to their nonuniform distribution, nonuniform quantization has to be used (see sublause Table 4.133).

After the LTP is determined, the predicted signal can be obtained for the $(m+1)$th frame

$$\hat{x}(i) = \sum_{k=-m_1}^{m_2} b_k \tilde{x}(i-1-k-\alpha),$$

$$i = mN+1, mN+2, \ldots, (m+1)N$$

Using the forward MDCT, we have the predicted spectral coefficients $\tilde{\mathbf{X}}_{m+1}$ for the $(m+1)$th frame. In order to guarantee that prediction is only used if it results in a coding gain, an appropriate predictor control is required and a

small amount of predictor control information has to be transmitted to the decoder. The predictor control scheme is the same as the backward predictor control scheme which has been used in MPEG-2 Advanced Audio Coding (AAC), i.e., one bit per scalefactor band is sent as side information to flag whether prediction is used for that band.

In the encoding process the prediction gain resulting from LTP is estimated for each scalefactor band by comparing the predicted energy to the amount of side information needed. LTP is switched on only for those scalefactor bands where there is positive net coding gain. The corresponding bits of side information are set accordingly. Finally, the net coding gain for the whole frame (all scalefactor bands) is checked, and the switch for using LTP at all for the frame is set accordingly.

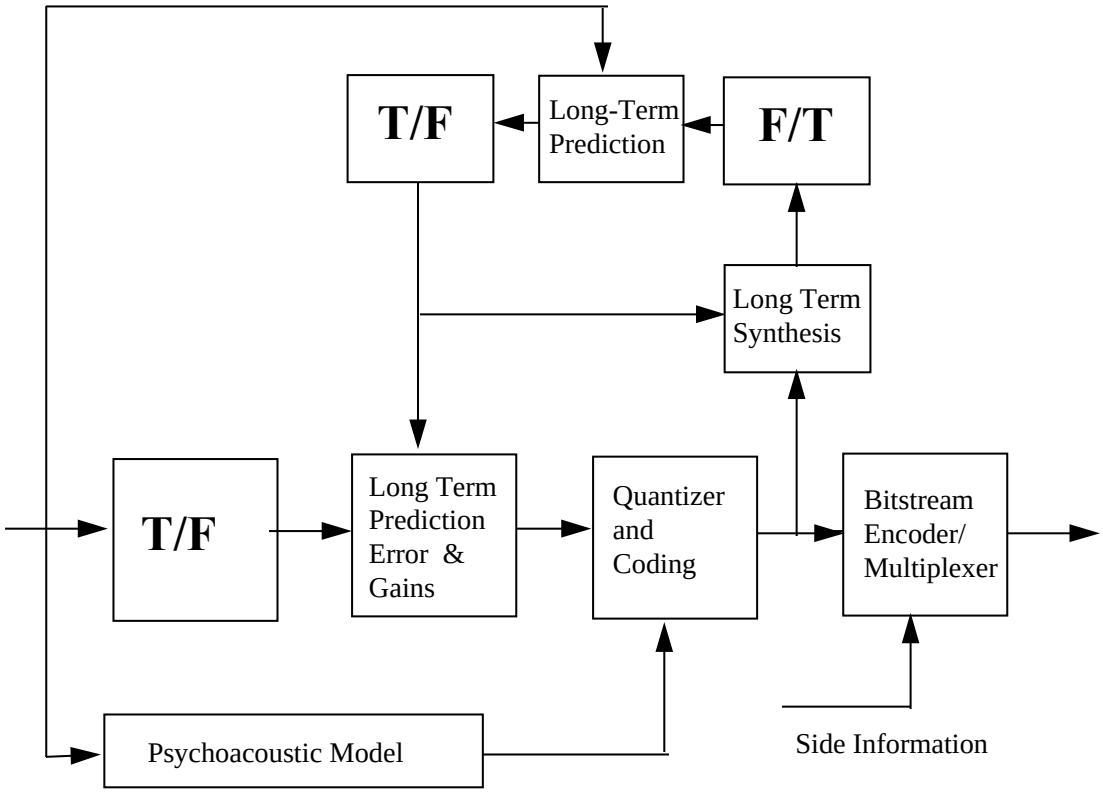


Figure 4.B.2 – Block diagram of the encoding process of Long Term Prediction

4.B.11 Perceptual Noise Substitution (PNS)

The encoding procedure for noise substitution is similar to the encoding procedure for intensity stereo and is performed as follows:

- For each scalefactor band containing spectral coefficients above a lower border frequency (e.g. 4 kHz) a noise detection is carried out. The scalefactor band is classified as noise-like if the corresponding signal is neither tonal nor contains strong changes in energy over time. The tonality of the signal can be estimated by using the tonality values calculated in the psychoacoustic model. Similarly, changes in signal energy can be evaluated using the FFT energies calculated in the psychoacoustic model.
- From the detection procedure, a map, noise_flag[sfb], is constructed such that noise-like scalefactor bands are flagged with a non-zero value.
- For each flagged scalefactor band the energy (sum of squares) of the corresponding spectral coefficients is calculated and mapped to a logarithmic representation with a resolution of 1.5 dB. An offset (NOISE_OFFSET=90) is added to the logarithmic noise energy values.
- For each flagged scalefactor band, the corresponding spectral coefficients are set to zero before quantization of the coefficients is carried out as usual.
- During the noiseless coding procedure, the pseudo-codebook NOISE_HCB is set for all flagged scalefactor bands. Apart from this, the regular section / noiseless coding procedure is carried out on the quantized coefficient data.

- The logarithmic noise energy values are coded analogous to the regular scalefactors, i.e. with a differential encoding scheme starting with the „global_gain value“. They are transmitted in place of the scalefactors belonging to the flagged scalefactor bands.

4.B.12 Random access points for GA coded bitstreams payloads

If neither the predictor nor the Long Term Predictor (LTP) is used in a bitstream payload the encoder should mark every frame as random accessible frame (on more details about random access see ISO/IEC 14496-1).

If the MPEG-2 style AAC predictor is used in a bitstream payload, none of the bitstream payload frames is randomly accessible. But nevertheless a decoder is allowed to tune into a stream of bitstream payload frames. For the MPEG-2 style AAC predictor it will take at maximum 240 frames until the full quality of the audio quality can be guaranteed (until each predictor reset group gets at least one reset).

If the MPEG-2 style AAC predictor is used in a bitstream payload, a frame is random accessible if the in non of the Predictor Reset Groups, the predictors are not used in the following frames until the corresponding Predictor Reset Group gets at least one reset.

Starting the decoder at a frame where LTP is on (and the previous frames are not available) can in some cases cause slight distortion. If evenly distributed entry points for random access without any possibility for distortion are desired, the LTP can just be switched off in the encoder for (at least) two frames every N^{th} frame. The Random Access Point (RAP) is then the first frame where LTP is off. This reset has no effect on the prediction gain in the next frames (no adaptation time). Therefore the number N can be small without significant effect on the average prediction gain. The same reset procedure can be used to stop propagation of frame loss errors when the LTP gain is very high. With very small values (<10) of N (number of frames between resets) a trade-off between average prediction gain and frame loss resilience can be made.

More complicated reset (switch-off) processes can be run in the encoder to optimise the bitstream payload for specific applications. For example the LTP flags for individual scalefactor bands can be used instead of the global LTP flag. In any case this is an encoder optimisation issue (and possibly error concealment issue in the decoder), and does not require any additional side information to be sent in the bitstream payload.

It should be noted that no predictor reset is needed with LTP for stability reasons.

Regarding error resilience, it should be noted that LTP and the signal buffer it requires can be used as an efficient tool for error concealment. Actual concealment methods are outside the scope of this standard.

4.B.13 Weighted interleave vector quantization

In the weighted interleave VQ section, the flattened MDCT coefficients vector and the weight vector is divided into subvector at the first stage. Then, the subvectors are applied to weighted vector quantization.

4.B.13.1 Initialization

Necessary parameters are initialized depending on the coding modes as described in subclause 4.6.5 of interleaved vector quantization tool.

4.B.13.2 Vector quantization

4.B.13.2.1 Basic structure

This vector quantizer has a two-channel conjugate structure, where two sets of codebooks are prepared. The search procedure consists of three steps; pre-selection, main selection and index packing.

4.B.13.2.2 Pre-selection

At the first step of the vector quantization procedure, candidates for the nearest codevector are selected from each codebook. To select the candidates, weighted distortion measures are calculated.

$$dist[idiv][icb] = \sum_{ismp=0}^{vec\_len[idiv]-1} wt_{sub}{}^{2}(sp\_cv0[icb][ismp] - U_{sub}[idiv][ismp])^{2},$$

$$\text{for } idiv = 0 \text{ to } N\_DIV - 1, icb = 0 \text{ to } 2^{bits[idiv]} - 1$$

If the *bits*0[*idiv*] is grater than MAXBIT_SHAPE, the following distortion are also calculated.

$$dist_{negative}[idiv][icb] = \sum_{ismp=0}^{vec\_len[idiv]-1} wt_{sub}{}^{2}(sp\_cv0[icb][ismp] + U_{sub}[idiv][ismp])^2,$$

$$\text{for } idiv = 0 \text{ to } N\_DIV - 1, icb = 0 \text{ to } 2^{bits[idiv]} - 1$$

The N_CAN candidates of dist and $dist_{negative}$ are selected with minimum distortion measure

4.B.13.2.3 Main selection

The distortion measures are calculated as below.

$$dist_{cross}[idiv][icb] = \sum_{ismp=0}^{vec\_len[idiv]-1} wt_{sub}[ismp] \left(\begin{array}{l} pol0[idiv] \\ \cdot sp\_cv0[can\_ind0[ican0]][ismp] \\ + pol1[idiv] \\ \cdot sp\_cv1[can\_ind1[ican1]][ismp] \end{array} \over 2 - U_{sub}[idiv][ismp] \right)$$

$$\text{for } idiv = 0 \text{ to } N\_DIV - 1, ican0 = 0 \text{ to } N\_CAN - 1, ican1 = 0 \text{ to } N\_CAN - 1$$

where

$$pol0[idiv] = 1 \quad \text{if } bits0[idiv] = MAXBIT\_SHAPE$$

$$pol0[idiv] = \begin{cases} 1 \\ -1 \end{cases} \quad \text{if } bits0[idiv] > MAXBIT\_SHAPE$$

$$pol1[idiv] = 1 \quad \text{if } bits1[idiv] = MAXBIT\_SHAPE$$

$$pol1[idiv] = \begin{cases} 1 \\ -1 \end{cases} \quad \text{if } bits1[idiv] > MAXBIT\_SHAPE$$

Indices *can_ind*0[*ican*0] and *can_ind*1[*ican*1] which give the least distortion measure is the quantization indices *index*0[*idiv*] and *index*1[*idiv*].

4.B.13.2.4 Index packing

For each VQ index, the polarity information is added as follows.

If $pol0[idiv] = -1$ then $index0[idiv] = index0[idiv] + 2^{MAXBIT\_SHAPE}$, for $idiv = 0$ to N_DIV

If $pol1[idiv] = -1$ then $index1[idiv] = index1[idiv] + 2^{MAXBIT\_SHAPE}$, for $idiv = 0$ to N_DIV

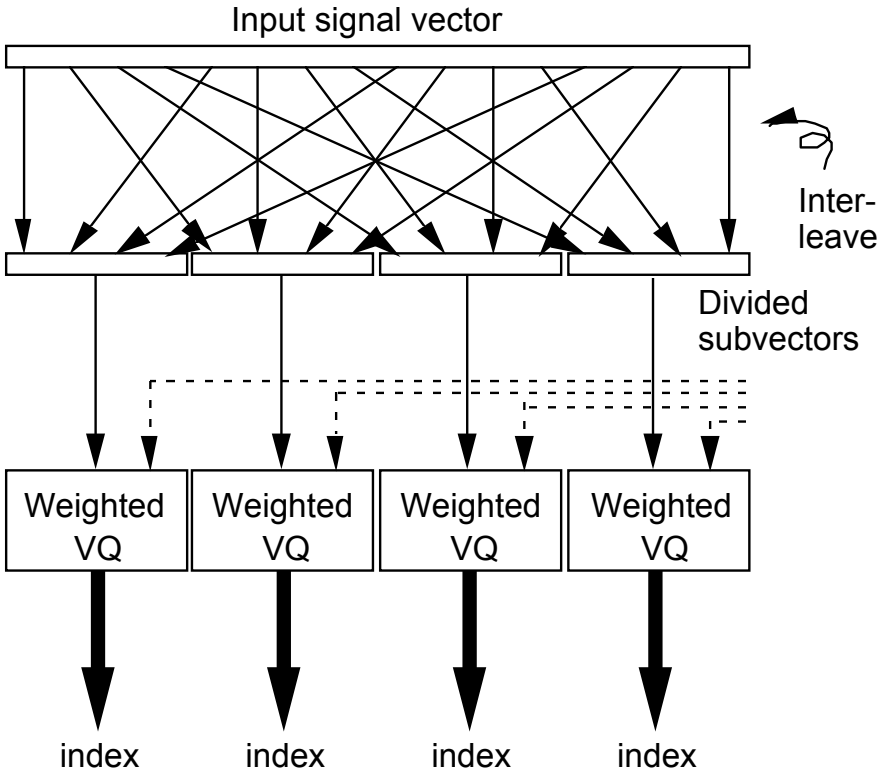


Figure 4.B.3 – Weighted interleave vector quantization.

4.B.14 Spectrum normalization

4.B.14.1 Encoding process

4.B.14.1.1 Bandwidth control

This process limits the bandwidth of MDCT spectrum for the quantization process.

4.B.14.1.2 LPC spectrum coding

At the first stage of the spectrum normalization process, input samples are divided into two paths. In one path they are transformed into frequency domain coefficients by MDCT. In the other path, LPC coefficients are calculated by LPC analysis; then, the LPC modeled spectrum envelope is calculated; and finally, each MDCT coefficient is divided by its corresponding spectrum envelope and first-stage flattened coefficients are produced. The main advantage of this flattening method is that fewer bits are required to normalize the frequency characteristics of input signals as a result of making use of an efficient quantization scheme of LSP (Line Spectral Pair) parameters. In order to avoid square root calculation in the de-flattening procedure at the decoder, LPC coefficients for the square root envelope are quantized; they are derived from the half-value of the LPC cepstrum of the normal LPC coefficients. The procedure for obtaining spectral envelope is listed as follows.

- - Autocorrelation function is calculated from the windowed input signal.
- - LPC coefficients are calculated by the Levinson-Durbin-Shur method.
- - LPC cepstrum coefficients are generated from LPC coefficients.
- - LPC cepstrum coefficients are multiplied by 0.5.
- - LPC coefficients (corresponding to square root spectrum) are converted from the LPC cepstrum coefficients
- by solving a normal equation.
- - Stability of the obtained coefficients is checked.
- - LPC coefficients are converted to LSP parameters.

LSP parameters are quantized by 2-stage split vector quantizer with moving average inter-frame prediction. Inverse LPC spectral envelope values are calculated by the same method in subclause 4.6.10 of the spectrum normalization tool

4.B.14.1.3 First-stage flattening

MDCT coefficients *X*[*isf*][*j*] are flattened using LPC spectrum as follows.

$$X_{flat1}[isf][ismp] = X[isf][ismp] / lpc\_spectrum[j],$$
$$\text{for } 0 \le isf < N\_SF,\ 0 \le ismp < N\_FR$$

4.B.14.1.4 Periodic peak components coding

If there are peak components with a constant period in the MDCT coefficients due to the harmonic nature of speech or some musical instruments in the long frame of the core coding mode, the process of the following periodic components coding is activated.

4.B.14.1.4.1 Fundamental frequency estimation

The best period value is searched that maximizes the energy of the periodic peak components. Number of extracted samples are fixed to 20 indifferent to the period. The definition of the period and the extracted samples are identical to those for the decoding process.

4.B.14.1.4.2 Vector quantization of the periodic peak components

The extracted peak components are normalized by the averaged amplitude and quantized by the interleaved weighted vector quantization similar to those used for flattened MDCT coefficients.

The averaged amplitude is quantized in a mu-law scale.

4.B.14.1.4.3 Subtraction of periodic peak components

The periodic peak components are locally decoded and subtracted from the original MDCT coefficients. The difference signals are used for the proceeding processes.

4.B.14.1.5 Bark-scale envelope coding

4.B.14.1.5.1 Calculation of the Bark-scale envelope

The inputs of the Bark-scale envelope calculation procedure are the MDCT coefficients xflat[] flattened by the LPC spectrum. First the square-rooted power of the input coefficients corresponding to each Bark-scale subband is calculated. The upper frequency boundary of each subband is described in subclause 4.6.10 of spectrum normalization tool.

$$env\_tmp[isf][ib] = \sqrt{\frac{\sum_{ismp=iblow}^{ibhigh-1} X_{flat1}[ismp]^2}{ibhigh - iblow}},$$
$$\text{for } 0 \le isf < N\_SF, 0 \le ib < N\_CRB$$

Next, the average of all square-rooted powers is calculated.

$$env\_avr[isf] = \frac{\sum_{ib=0}^{N\_CRB-1} env\_tmp[isf][ib]}{N\_CRB}, \quad \text{for } 0 \le isf < N\_SF$$

Then, square-rooted powers are normalized by the average to create the MDCT envelope.

$$env[isf][ib] = env\_tmp[isf][ib] / env\_avr[isf],$$
$$\text{for } 0 \le isf < N\_SF, 0 \le ib < N\_CRB$$

4.B.14.1.5.2 Weight calculation

The weight is used for the vector quantization of the MDCT envelope.

$$env\_fwt[ib] = \frac{\sum_{ismp=iblow}^{ibhigh-1}(lpc\_spectrum[ismp])}{ibhigh - iblow}$$

4.B.14.1.5.3 Quantization

4.B.14.1.5.3.1.1 Interframe prediction

Before quantizing the MDCT envelope, the prediction switch is set. The values of envelope elements are subtracted by their average (is 1).

$$env[is][ib] = env[isf][ib] - 1, \quad \text{for } 0 \le isf < N\_SF, 0 \le ib < N\_CRB$$

Correlation between current-frame env[][] and previous-frame env[][] is calculated.

$$correlation = \frac{\sum_{ib=0}^{N\_CRB-1} env_{current}[isf][ib] \cdot env_{previous}[isf][ib]}{\sum_{ib=0}^{N\_CRB-1}(env_{current}[isf][ib])^2}, \quad \text{for } 0 \le isf < N\_SF$$

if N_SF is not equal to 1, the envelope of the previous frame is set by a following equation:

$$env_{previous}[0][ib] = 0,$$
$$env_{previous}[isf][ib] = env_{current}[isf-1][ib], \quad \text{for } 0 \le isf < N\_SF, 0 \le ib < N\_CRB$$

If the correlation measure is greater than 0.5, prediction is on, otherwise, prediction is off.

$$\text{if } (correlation[isf] > 0.5) \text{ then } index\_fw\_alf[isf] = 1$$
$$\text{else} \qquad index\_fw\_alf[isf] = 0,$$
$$\text{for } 0 \le isf < N\_SF$$

4.B.14.1.5.3.1.2 Vector quantization of the MDCT envelope

At the first step of quantization, the envelope and the weight vector are divided into FW_N_DIV subvectors.

$$denv[isf][ifdiv][icv] = env[isf][FW\_N\_DIV * icv + ifdiv],$$
$$\text{for } 0 \le isf < N\_SF, 0 \le ifdiv < N\_CRB, 0 \le icv < FW\_CB\_LEN$$

$$dfwt[ifdiv][icv] = env\_fwt[FW\_N\_DIV * icv + ifdiv],$$
$$\text{for } ifdiv = 0 \text{ to } N\_CRB - 1, icv = 0 \text{ to } FW\_CB\_LEN - 1$$

Distortion measures $fwdist[itmp]$ are calculated as follows.

$$alfq[isf] = index\_fw\_alf[isf] \cdot FW\_ALF\_STEP$$

$$fwdist[isf][itmp] =$$
$$\sum_{icv=0}^{FW\_CB\_LEN-1} dfwt[ifdiv][icv]^2 \cdot$$
$$(denv[isf][ifdiv][icv] - alfq \cdot p\_cv\_env[isf][icv] - cb\_env[itmp][icv])^2$$
$$\text{for } 0 \le isf < N\_SF, 0 \le itmp < FW\_CB\_SIZE, 0 \le ifdiv < FW\_N\_DIV$$

The cv_env[][] are elements of the envelope codebook. The p_cv_env[] represents the MDCT envelope of the previous frame. It is calculated by the same procedure mensioned in subclause 4.6.10 of spectrum normalization tool.

Finally, $itmp$ for the minimum distortion measure $fwdist[itmp]$ is the quantization index $index\_env[ifdiv]$.

4.B.14.1.5.3.1.3 Local decoding

After the vector quantization, the MDCT envelope is reproduced as follows.

```
for (isf = 0; isf < N_SF; isf++) {
   alfq[isf] = index_fw_alf[isf] * FW_ALF_STEP
   for (ifdiv = 0; ifdiv < FW_N_DIV; ifdiv++) {
      for (icv = 0; icv < FW_CB_LEN; icv++) {
         ienv = FW_N_DIV * icv + ifdiv;
         dtmp = cv_env[index_env[isf][ifdiv]][icv];
         qenv_b[isf][ienv] = dtmp + alfq * p_cv_env[isf][icv] + 1;
         if (N_SF == 1)
             p_cv_env[isf][icv] = dtmp;
         else
                 p_cv_env[isf-1][icv] = dtmp;
      }
   }
}
```

The cv_env[][] are the envelope codebook.

Then, Bark-scale envelope *qenv_b*[*isf*][*ienv*] are projected into linear-scale envelope *qenv*[*isf*][*ismp*].

```
for (isf = 0; isf < N_SF; ist++) {
   for (ib = 0; ib < N_CRB; ib++) {
      for (ismp = iblow[ib]; ismp < ibhigh[ib]; ismp++) {
         qenv[isf][ismp] = qenv_b[isf][ib];
      }
   }
}
```

4.B.14.1.6 Second-stage flattening

X_{flat2}[][] are flattened as follows:

$$X_{flat2}[isf][j] = X_{flat1}[isf][j] / qenv[j] \qquad \text{for } 0 \le isf < N\_SF, 0 \le j < N\_FR$$

4.B.14.1.7 Gain quantization and amplitude normalization

Before quantizing the flattened MDCT coefficients, their amplitudes are normalized by the gain factor.

The gain factor is calculated as follow:

$$gain[isf] = \frac{\sqrt{\frac{\sum_{ismp=0}^{N\_FR-1} X_{flat2}[isf]^2}{N\_FR}}}{AMP\_NM}, \quad \text{for } 0 \le isf < N\_SF$$

If N_SF == 1, the gain factor is quantized using the following equation:

$$index\_gain[0] = (\text{int}) \frac{\log 10(1 + MU \cdot gain[0] / AMP\_MAX)}{\log 10(1 + MU)}$$

Then, gain factor is locally decoded.

g_temp = index_gain * STEP + STEP / 2

qgain = AMP_MAX * (exp10(g_temp * log10(1.+MU) / AMP_MAX) -1) / MU

qgain /= AMP_NM

If N_SF > 1, the global gain and subframe gains are quantized:

$$index\_gain[0] = (\text{int})\frac{\log 10(1 + MU \cdot gain[0] / AMP\_MAX)}{\log 10(1 + MU)}$$

Finally, the flattened MDCT coefficients are normalized by the decoded gain factor *qgain*.

$$U[ismp] = X_{flat2}[ismp] / qgain \quad \text{for } ismp = 0 \text{ to } N\_FR - 1$$

4.B.15 Scalable AAC with core coder

Mono or Stereo Encoder

The description provided below describes one possible way of achieving bit rate scalability. Bit-rate functionality can also be achieved using individual coding schemes as well. Here, a particular configuration is described where the GA modules and a core coder is used. Since it is based on the calculation of a difference signal, the core coder must encode the waveform of the input signal. The MPEG-4 CELP coder is the coder to be used with this tool in MPEG-4 Audio. Furthermore, the tool has been successfully tested with the ITU-T G.729 and G.723.1 codec and with the FS1016 CELP coder. However none of theses coders is available in a MPEG-4 system.

In addition to the core coder there is at least one enhancement layer based on the MPEG-4 Audio GA coding modules. In order to allow for integer frame lengths, depending on the core coder, alternative frame lengths for the AAC module may be used. The GA modules provide, in addition to the standard 1024 length, also a 960 samples per frame implementation, which at 48 kHz sampling rate leads to 20 ms and at 32 kHz to a frame length of 30 ms. This allows for an easy integration of the MPEG-4 CELP core, and to construct bitstream payload frames which integrate standard speech coders, like G.729, which have a frame length of a multiple of 10 ms.

The bitrate of the additional layers can be any bit rate defined for the GA based enhancement stages. The ratio of the sampling rate of the core coder and the sampling rate of the enhancement coder must be an integer.

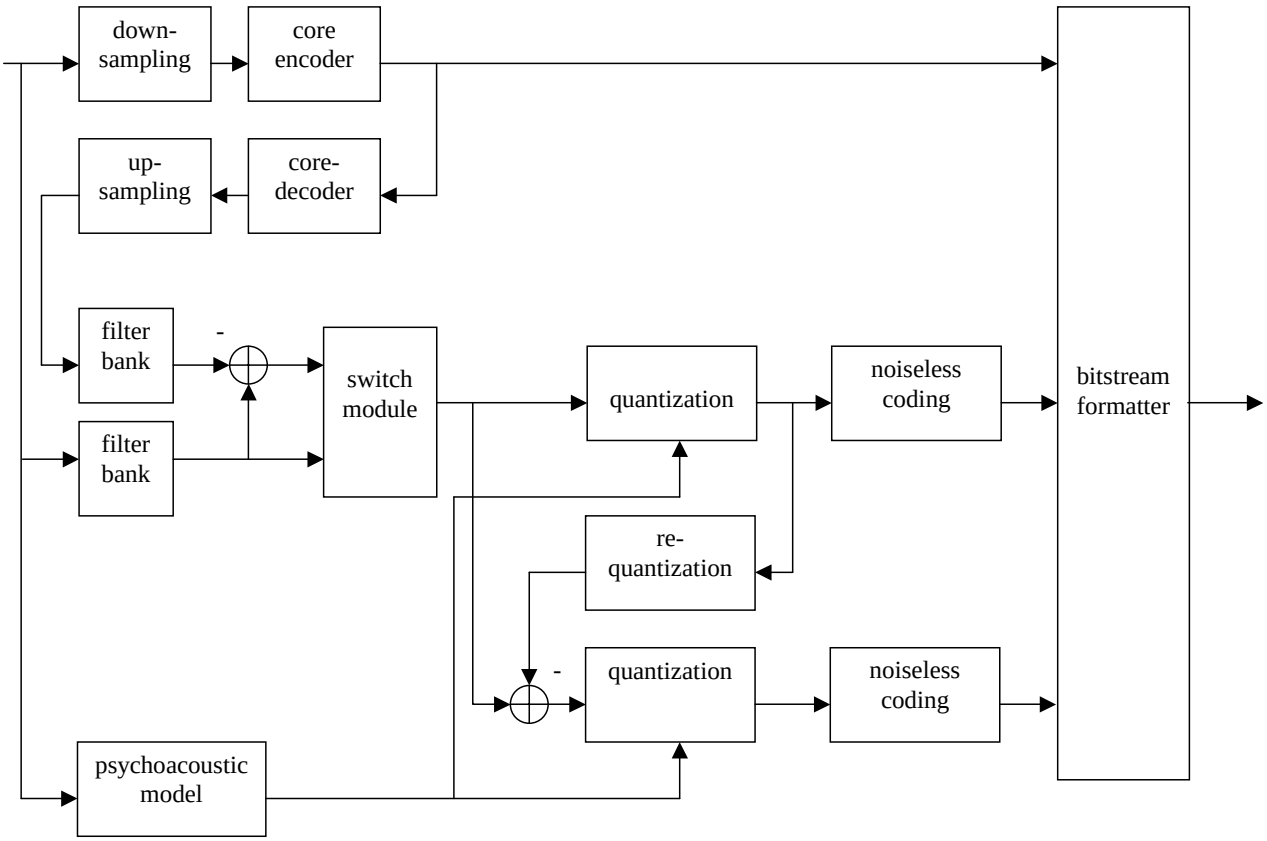


Figure 4.B.4 – Encoder process

Mixed Mono/Stereo Coder

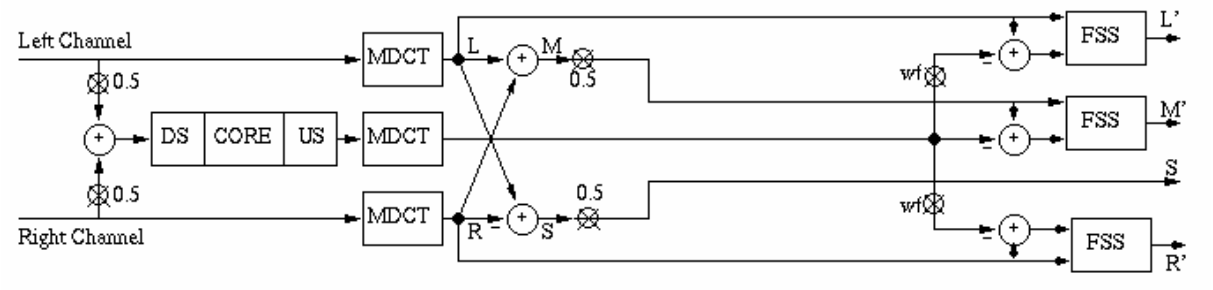


Figure 4.B.5 – Core + stereo GA

From the stereo input signal a mono signal is derived, which is then coded/decoded with a core coder, which - in general - operates at a lower sampling rate, up-sampled and transformed into the frequency-domain. This part is identical to the corresponding part of the mono scalable coder. Next the standard M/S-Stereo matrix, as defined for the AAC coder is calculated on the spectra of the Left (L) and Right (R) signal, giving the Mid (M) and Side (S) signals. Three Frequency-Selective Switch modules (FSS) - identical to the switch modules of the mono scalable coder - then are used to generate the signals L', R' and M' from the L, R, and M signals. These signals and the original S signal then are used as the input signal to the standard AAC joint stereo coding modules. Both, M/S- and Intensity stereo coding is possible. The standard AAC ms_mask_present and ms_used[] flags are used to indicate M/S or L/R coding. However, since either M/S or L/R coding is used, it is not necessary to transmit all three FSS-related side-information. Instead, if M/S coding is selected for a band, only the FSS information for the M-switch needs to be transmitted in diff_control[0][win][sfb]. diff_control[1][win][sfb] is not transmitted in this case. If L/R coding is selected the FSS-side-information for the L and R channel needs to be transmitted in diff_control[0][win][sfb] for the left channel and in diff_control[1][win][sfb] for the right channel.

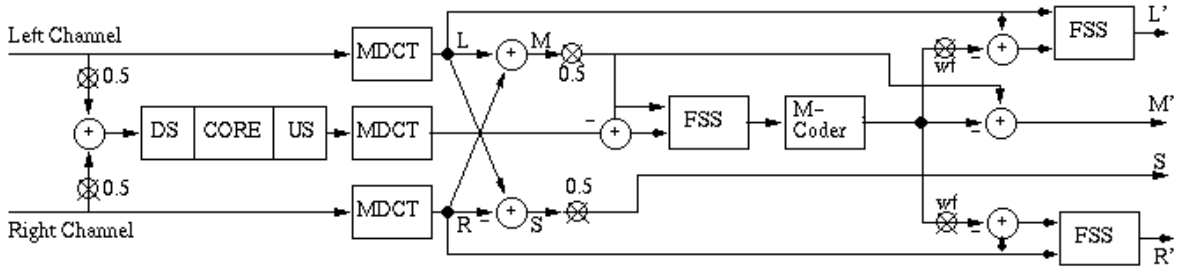


Figure 4.B.6 – Core + mono GA + stereo GA

This is an extended version of the Core + stereo GA coder. An additional GA - Coder is used to encode the output signal of the core-coder in the frequency domain. This makes the core plus M - GA coder combination identical to the scalable mono core plus GA coder. Two more FSS modules allow to select L'/R' to be identical to L, or R, respectively, or L - M*wf or R-M*wf. Again some of the switching conditions need not to be transmitted: If M/S coding is selected , only the FSS information for the M-switch needs to be transmitted in diff_control[0][win][sfb]. Diff_control[1][win][sfb] is not transmitted.. If L/R coding is selected the FSS-side-information for the L and R and the M channel needs to be transmitted in diff_control[0][win][sfb] for the left channel and in diff_control[1][win][sfb] for the right channel, and diff_control_m[win][0] for the M channel.

4.B.16 Scalable controller

This tool controlls the spectrum normalization and interleave vector quantization tools to construct a scalable coder. In each scalable layer, spectrum normalization and interleaved vector quantization tool is cascaded. Each scalable layer has active frequency band which is shifted according to signal characteristics.

4.B.17 Fine grain scalability: BSAC (Bit-Sliced Arithmetic Coding)

4.B.17.1 Introduction

In the BSAC encoder the inputs to the noiseless coding module are the set of 1024 quantized spectral coefficients and the scalefactor of the scalefactor band. Since the noiseless coding is done inside the quantizer inner loop, it is part of an iterative process that converges when the total bit count (of which the noiseless coding is the vast majority) is within some interval surrounding the allocated bit count. This subclause will describe the encoding process for a single call to the noiseless coding module.

Noiseless coding is done via the following steps:

- Bit Slicing of the quantized spectral coefficient
- Preliminary Arithmetic coding of the set of qunatized spectum within a coding band using the probability table
- Preliminary Arithmetic coding of the scalefactors, stereo infomation, artihmetic model infomation.
- Probability table determination to achieve lowest bit count

4.B.17.2 Bit Slicing of the quantized spectral coefficients

As a first step lof BSAC encoding process, a sequence of the absolute values of quantized spectral coefficients is mapped into a bit-sliced sequence as shown in the following Figure 4.B.7.

MSB			LSB	
$B_{0,m}$	$B_{0,m-1}$	...	$B_{0,0}$	0^{th} quantized spectral data
$B_{1,m}$	$B_{1,m-1}$	...	$B_{1,0}$	
$B_{2,m}$	$B_{2,m-1}$	...	$B_{2,0}$	
...	...	...	...	
$B_{k,m}$	$B_{k,m-1}$	...	$B_{k,0}$	k^{th} quantized spectral data

where, MSB plane is (m+1) bit and $B_{k,m}$ indicates the binary value of the m^{th} bit-slice of k^{th} quantized spectral coefficients

Figure 4.B.7 – Bit slicing of the quantized spectral coefficients

For example, consider a sequence of the absolute values, x[n] as follows :

x[0] = 9, x[1]=0, x[2]=7 and x[3]=11 ...

If MSB plane is 5, bit-slices are formed from a quantized sequence as shown in Figure 4.B.8 :

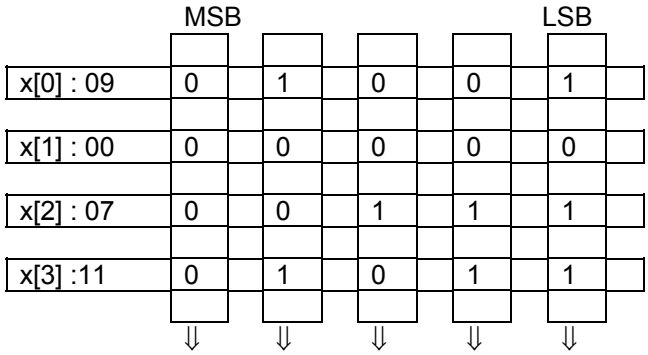


Figure 4.B.8 – Example of Bit-Slicing

4.B.17.3 Probability table determinations

Several probability tables are provided to cover the different statistics of the bit-slices. One probability table is used for encoding the bit-sliced data of each coding band. The noiseless coding segments the set of 1024 quantized spectral coefficients into *coding bands*, such that a single probability table is used to code each coding band (the method of Arithmetic coding is explained in a later subclause). For reasons of coding efficiency, the quantized spectral coefficients are divided into *coding bands* which contain 32 quantized spectral coefficients for the noiseless coding. *Coding bands* are the basic units used for the noiseless coding for BSAC.

The bit-sliced data is decoded with the probability value which is selected among the values of the BSAC probabiliy table.

The probability value should be defined in order to arithmetic-code the symbols (the sliced bits). Binary probability table is made up of probability values (p0) of the symbol '0'. First of all, sub-table is selected according to the significance and the coded higher bits of the quantized spectra. The offset for the probability (p0) can be decided using the sliced bits of successive non-overlapping 4 spectral data in order to select one of the several probability values in the sub-table. However if the available codeword size is smaller than 14, there is a constraints on the selected probability value.

Probability table index is determined among the possible tables, such that the number of bits needed to represent the full set of the bit-sliced data of quantized spectral coefficients within each coding band is minimized. The possible arithmetic models have the number of the allocated bit larger than or equal to that of the bit needed to represent the PCM data of quantized spectral coefficients within a coding band.

Coding bands often contain only coefficients whose value is zero. For example, if the audio input is band limited to 20 kHz or lower, then the highest coefficients are zero. Such coding bands are coded with probability table 0, where the allocated bit is 0 and all coefficients are zero.

In order to transmit the probability table information used in encoding process, it is included the coding band side information (cband_si) and coded in the syntax of layer_cband_si(). The probability table index for encoding the bit-sliced data within each coding band is transmitted starting from the lowest frequency coding band and progressing to the highest frequency coding band. For all arithmetic model indices the value is arithmetic-coded. After the model index is encoded, the encoding of the bit-sliced data shall be started.

4.B.17.4 Grouping and interleaving

If the window sequence is eight short windows then the set of 1024 coefficients is actually a matrix of 8 by 128 frequency coefficients representing the time-frequency evolution of the signal over the duration of the eight short windows. Although the sectioning mechanism is flexible enough to efficiently represent the 8 zero sections, *grouping* and *interleaving* provide for greater coding efficiency. As explained earlier, the coefficients associated with contiguous short windows can be grouped such that they share scalefactors amongst all scalefactor bands within the group. In addition, the coefficients within a group are interleaved by interchanging the order of scalefactor bands and windows. To be specific, assume that before interleaving the set of 1024 coefficients *c* are indexed as

c[g][w][k/4][k%4]

where

g is the index on groups,

w is the index on windows within a group,

k is the index on coefficients within a window,

and the right-most index varies most rapidly.

After interleaving the coefficients are indexed as

c[g][k/4][w][k%4]

This has the advantage of combining all zero sections due to band-limiting within each group.

4.B.17.5 Scalefactors

The coded spectrum uses one quantizer per scalefactor band. The step size of each of these quantizers is specified as a set of scalefactors and a maximum scalefactor that normalizes these scalefactors. In order to increase compression, scalefactors associated with scalefactor bands that have only zero-valued coefficients are ignored in the coding process and therefore do not have to be transmitted. Both the maximum scalefactor and scalefactors are quantized in 1.5 dB steps.

The BSAC scalable coding scheme includes the noiseless coding to reduce the redundancy of the scalefactors. The maximum scalefactor is coded as an 8 bit unsigned integer. The first scalefactor associated with the quantized spectrum is differentially coded relative to the maximum scalefactor and the arithmetic coded using the differential scalefactor arithmetic model. The remaining scalefactors are differentially coded relative to the previously encoded scalefactor and then Arithmetic coded using the differential scalefactor model. The dynamic range of the maximum scalefactor is sufficient to represent full-scale values from a 24-bit PCM audio source.

4.B.17.6 Arithmetic coding

BSAC uses the bit-slicing scheme of the quantized spectral coefficients in order to provide the fine grain scalability. And it encode the bit-sliced data using binary arithmetic coding scheme in order to reduce the average bits transmitted while suffering no loss of fidelity.

In BSAC scalable coding scheme, a quantized sequence is divided into coding bands. And, a quantized sequence is mapped into a bit-sliced sequence within a coding band. The noiseless coding of the sliced bits relies on probability table of the coding band, the significance and the other contexts.

The significance of the bit-sliced data is the position of the sliced bit to be coded.

The flags, sign_is_coded[] are updated with coding the vectors from MSB to LSB. They are initialized to 0. And they are set to 1 when the sign of the quantized spectrum is coded.

The probability table for encoding the bit-sliced data within each coding band is included in the bistream element **cband_si_type** and transmitted starting from the lowest coding band and progressing to the highest coding band allocated to each layer.

The length of the available bitstream payload (*available_len[]*) is initialized at the beginning of each layer. The estimated length of the codeword (*est_cw_len*) to be decoded is calculated from the arithmetic decoding process. After the arithmetic encoding of a symbol, the length of the available bitstream payload should be updated by subtracting the estimated codeword length from it. We can detect whether the remaing bitstream payload of each layer is available or not by checking the available_len.

The sign bits associated with non-zero coefficients follow the arithmetic codeword when the bit-value of the quantized spectral coefficient is 1 for the first time, with a 1 indicating a negative coefficient and a 0 indicating a positive one. The flag, *sign_is_coded[]* represents whether the sign bit of the quantized spectrum has been decoded or not. When the flag, *sign_is_coded* is 0 and the bit value of quantized spectral coefficient is nonzero, the sign bit of a sample is binary arithmetic encoded. The flag, *sign_is_coded* is set to 1 after the sign bit is encoded.

4.B.17.6.1 Arithmetic coding procedure

Arithmetic Coding consists of the following 3 steps :

- Initialization which is performed prior to the coding of the first symbol
- Coding of the symbol themselves.
- Termination which is performed after the decoding of the last symbol

4.B.17.6.1.1 Registers, symbols and constants

Several registers, symbols and constants are defined to describe the arithmetic encoder.

- *coding_mode* : 1-bit fixed point register which represents the 1-bit scale-up is used or not.
- *half*: 32-bit fixed point constant equal to 1/2(0x20000000)
- *qtr*: 32-bit fixed point constant equal to 1/4(0x10000000)
- *qtr3*: 32-bit fixed point constant equal to 3/4(0x30000000)
- *low*: 32-bit fixed point register. Contains the lower bound of the interval
- *range*: 32-bit fixed point register. Contains the range of the interval.
- *p0*: 16-bit fixed point register. Probability of the “0” symbol.
- *p1*: 16-bit fixed point register. Probability of the “1” symbol.
- *cum_freq* : 16-bit fixed point registers. Cummulative Probabilities of the symbols.

4.B.17.6.1.2 Initialization

The lower bound *low* is set to 0, the range *range* to *half**2 (0x40000000).

4.B.17.6.1.3 Encoding a symbol

Arithmetic encoding procedure varies on the symbol to be encode. If the symbol is the sliced bit of the spectral data, the binary arithmetic encoding is used. Otherwise, the general arithmetic encoding is used.

When a symbol is binary arithmetic-encoded, the probability p0 of the “0” symbol is provided according to the context computed properly and using the probability table. p0 uses a 6-bit fixed-point number representation. Since the decoder is binary, the probability of the “1” symbol is defined to be 1 minus the probability of the “0” symbol, i.e. p1 = 1-p0.

When a symbol is encoded using arithmetic encoding, the cummulative probability values of multiple symbols are provided. The probability values are regarded as the arithmetic model. The arithmetic model for encoding a symbol is transmitted in the data elements. For example, arithmetic models of scalefactor and cband_si are transmitted in the data elements, **base_scf_model**, **enh_scf_model** and **cband_si_type**. Each value of the arithmetic model uses a 14-bit fixed-point representation.

4.B.17.6.1.4 Termination

After the last symbol has been coded in the arithmetic encoder, additional bits need to be “introduced” to guarantee decodability.

4.B.17.7 Stereo-related data and PNS data

The BSAC scalable coding scheme includes the noiseless coding which is different from MPEG-4 AAC coding and further reduce the redundancy of the stereo-related data. Encoding of the stereo-related data and Perceptual Noise Substitution(pns) data is depended on pns_data_present and **stereo_info** which indicates the stereo mask. Since the decoded data is the same value with MPEG-4 AAC, the MPEG-4 AAC stereo-related data and pns processing follows the decoding of the stereo-related data and pns data. The detailed encoding process of stereo-related data and pns data is classified as follows:

- 1 channel, no pns data: If the number of channel is 1 and pns data is not present, we do not need data elements related to stereo or pns.
- 1 channel, pns data: If the number of channel is 1 and pns data is present, noise flag of the scalefactor bands between **pns_start_sfb** to **max_sfb** is arithmetic encoded. Perceptual noise substitution is done according to the noise flag.
- 2 channel, ms_mask_present = 0 (Independent), No pns data: If ms_mask_present is 0 and pns data is not present, arithmetic decoding of stereo_info or ms_used is not needed.
- 2 channel, ms_mask_present = 0 (Independent), pns data: If ms_mask_present is 0 and pns data is present, noise flag for pns is arithmetic encoded. Perceptual noise substitution of independent mode is done according to the noise flag.
- 2 channel, ms_mask_present = 2 (all ms_used), pns data or no pns data: All ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band. And naturally there can be no pns processing regardless of pns_data_present flag.
- 2 channel, ms_mask_present = 1 (optional ms_used), pns data or no pns data: One bit mask of ms_used per band for max_sfb bands is conveyed in this case. ms_used is arithmetic encoded using the ms_used model. M/S stereo processing of AAC is done or not according to the decoded ms_used. And there is no pns processing regardless of pns_data_present flag
- 2 channel, ms_mask_present = 3 (optional ms_used/intensity/pns), no pns data: At first, stereo_info is arithmetic encoded using the stereo_info model. stereo_info is is two-bit flag per scalefactor band indicating that M/S coding or Intensity coding is being used in window group g and scalefactor band sfb as follows :

  00 independent

  01 ms_used

  10 intensity_in_phase

  11 intensity_out_of_phase

- If ms_mask_present is not 0, M/S stereo or intensity stereo of AAC is done with these data. Since pns data is not present, we do not have to process pns.

2 channel, ms_mask_present = 3 (optional ms_used/intensity/pns), pns data: stereo_info is arithmetic encoded using the stereo_info model. If stereo_info is 1 or 2, M/S stereo or intensity stereo processing of AAC is done with these data and there is no pns processing. If stereo_info is 3 and scalefactor band is larger than or equal to pns_start_sfb, noise flag for pns is arithmetic encoded. And then if the both noise flags of two channel are 1, noise substitution mode is arithmetic encoded. Otherwise, the perceptual noise is substituted only if noise flag is 1.

4.B.17.8 Payload transmitted over elementary stream

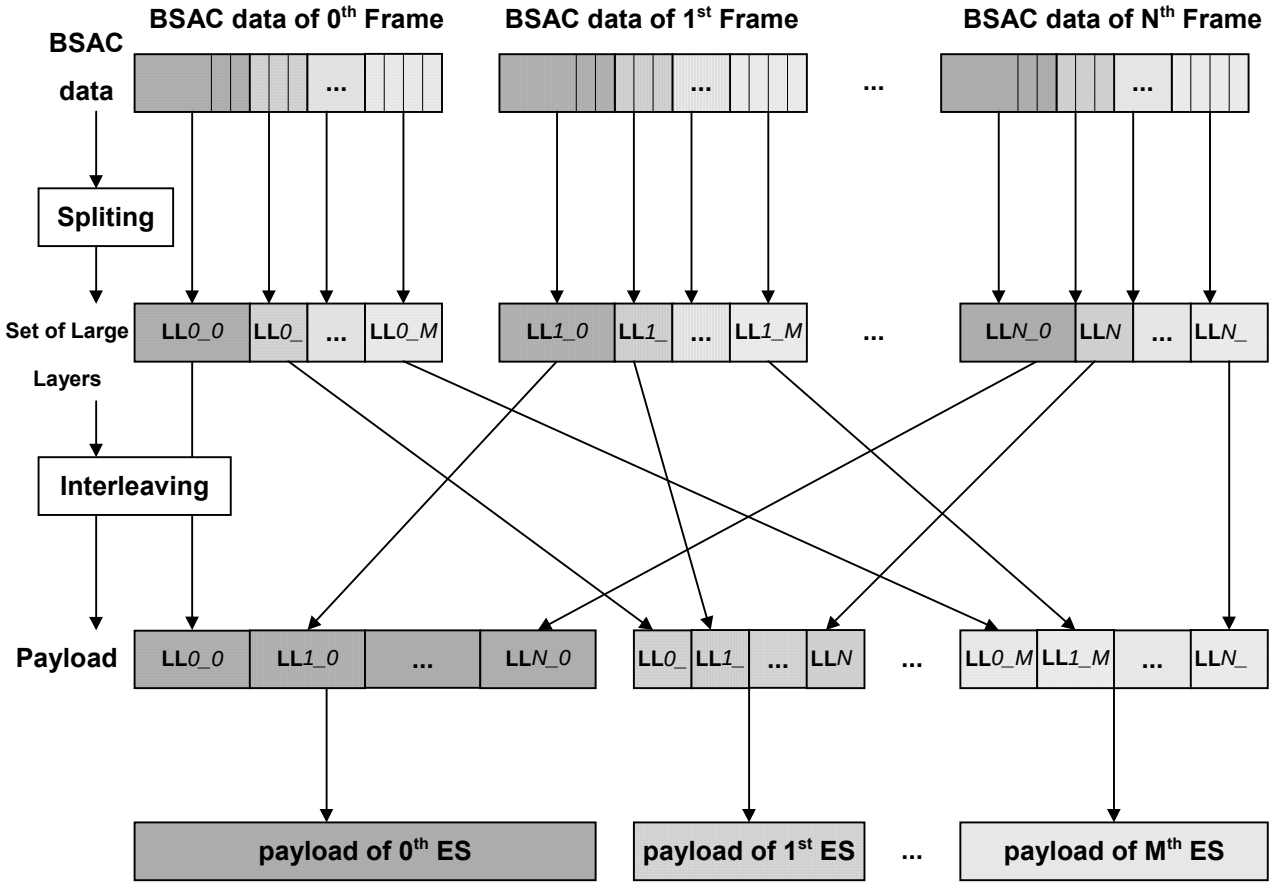
Fine grain scalability would create large overhead if one would try to transmit fine step scalability over multiple elementary streams. So, the server can organize the BSAC data into the payloads in order to reduce overhead and implement the fine step scalability efficiently in MPEG-4 system. The configuration of the payload can be changed according to the environment such as the user interaction or the network traffic. The organization scheme is as follows:

- BSAC data of a frame is split into the large-step layers for transmission. The number of the layer and the length of each layer depend on the application that the content creator is willing to provide to the users. The length of

the large-step layer, *layer_length[]* can be calculated accoring to the bitrate of the layer to be transmitted, the sampling frequency and the frame length in sample. The length of the large-step layer is transmitted to the receiver through the syntax element **layer_length** in GA Specific Configuration. In the course of the transmission, this value can be changed at a request of bsac_backchan_stream(). And if the user request the configuration of the payload throgh the back-channel, these values can be changed according to the syntax elements **numOfLayer** and **Avg_bitrate** of bsac_backchan_stream().

- BSAC data of several frames are grouped and transmitted at a time. The layers of the sub-frames are interleaved into the payload. The number of the grouped frame depends on the available delay of the application that will be transmitted to the users. The content provider should initialize this number to the proper value. And if the user request the configuration of the payload throgh the backchannel, this value can be changed. This number is transmitted to the receiver in the value of the syntax element **noOfSubFrame** in GA Specific Configuration. In the course of the transmission, this value can be changed at a request of bsac_backchan_stream().

In the server, the payload is formed as shown in the following Figure 4.B.9.



where, **LL*i*_*k*** is the k-th large-step layer of the i-th sub-frame

(M+1) is the number of the large-step layer to be transmitted (numOfLayer)

(N+1) is the number of the sub-frame to be grouped in an AU (numOfSubFrame)

Figure 4.B.9 – The manufacture process of the BSAC payload

1. The BSAC data of i-th sub-frame is split into several large layers for transmission over ES as in Figure 4.B.10.

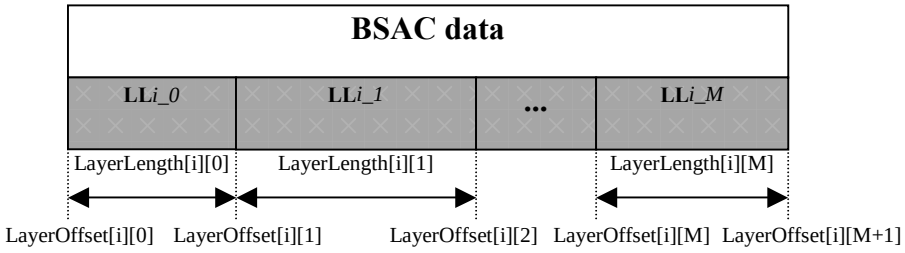


Figure 4.B.10 – The structure of the BSAC data

2. Large layers of M-subsequent frames are interleaved and concatenated to the payloads.

Some help variables and arrays are needed to describe the generation process of the payload transmitted over ES. These help variables depend on layer, numOfLayer, numOfSubFrame, **layer_length** and **frame_length** and must be built up for mapping bsac_raw_data_block() of each sub-frame into the payloads. The pseudo code shown below describes

- how to calculate *LayerLength[i][k]*, the length of the large-step layer which is located on the fine granule audio data, bsac_raw_data_block() of i^{th} sub-frame.
- how to caluculate *LayerOffset[i][k]* which indicates the start position of the large-step layer of i^{th} frame which is located on the payload of the k^{th} ES (bsac_payload())
- how to calculate *LayerStartByte[i][k]* which indicates the start position of the large-step layer which is located on the fine granule audio data, bsac_raw_data_block() of i^{th} sub-frame

```
for (k = 0; k < numOfLayer; k ++) {
   LayerStartByte[0][k] = 0;
   for (i = 0; i < numOfSubFrame; i++) {
      if (k == (numOfLayer-1) ) {
         LayerEndByte[i][k] = frame_length[i];
      } else {
      LayerEndByte[i][k]=LayerStartByte[i][k] + layer_length[k];
      if (frame_length[i] < LayerEndByte[i][k])
         LayerEndByte [i][k] = frame_length[i];
      }
      LayerStartByte[i+1][k] = LayerEndByte[i][k];
      LayerLength[i][k] = LayerEndByte[i][k] - LayerStartByte[i][k];
   }
}
for (k = 0; k < numOfLayer; k ++) {
   LayerOffset[0][k] = 0;
   for (i = 0; i < numOfSubFrame; i++) {
      LayerOffset[i+1][k] = LayerOffset[i][k] + LayerLength[i][k]
   }
}
```

Where *frame_length[i]* is the length of i^{th} frame bitstream payload which is obtained from the syntax element **frame_length** and layer_length[i] is the average length of the large-step layers in the payload of i^{th} layer ES.

4.B.18 Informative SBR encoder description

4.B.18.1 Encoder overview

The encoder part of the SBR tool estimates several parameters used by the high frequency reconstruction method on the decoder. The basic layout is depicted below.

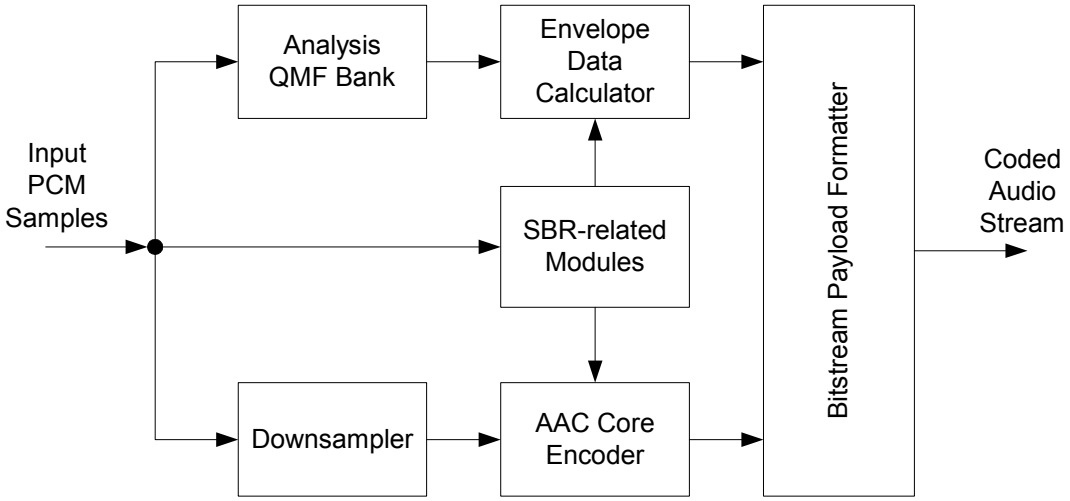


Figure 4.B.11 – Encoder overview

4.B.18.2 Analysis filterbank

Subband filtering of the input signal is done by a 64-subband QMF bank. The output from the filterbank, i.e. the subband samples, are complex-valued and thus oversampled by a factor two compared to a regular QMF bank. The flowchart of this operation is given in Figure 4.B.12. The filtering comprises the following steps, where an array $\mathbf{x}$ consisting of 640 time domain input samples are assumed. Higher indices into the array corresponds to older samples:

- Shift the samples in the array $\mathbf{x}$ by 64 positions. The oldest 64 samples are discarded and 64 new samples are stored in positions 0 to 63.
- Multiply the samples of array $\mathbf{x}$ by window $\mathbf{c}$. The window coefficients are found in Table 4.A.87.
- Sum the samples according to the formula in the flowchart to create the 128-element array $\mathbf{u}$.
- Calculate 64 new subband samples by the matrix operation $\mathbf{Mu}$, where

$$\mathrm{M}(\mathrm{k,n}) = \exp\left(\frac{i \cdot \pi \cdot (k + 0.5) \cdot (2 \cdot n + 1)}{128}\right), \begin{cases} 0 \le k < 64 \\ 0 \le n < 128 \end{cases}$$

  In the equation, $\exp()$ denotes the complex exponential function and i is the imaginary unit.

Every loop in the flowchart produces 64 complex-valued subband samples, each representing the output from one filterbank subband. For every SBR frame the filterbank will produce $numTimeSlots \cdot RATE$ subband samples from every filterbank subband, corresponding to a time domain signal of length $numTimeSlots \cdot RATE \cdot 64$ samples. In the flowchart $\mathbf{X}$[k][l] corresponds to subband sample l in QMF subband k.

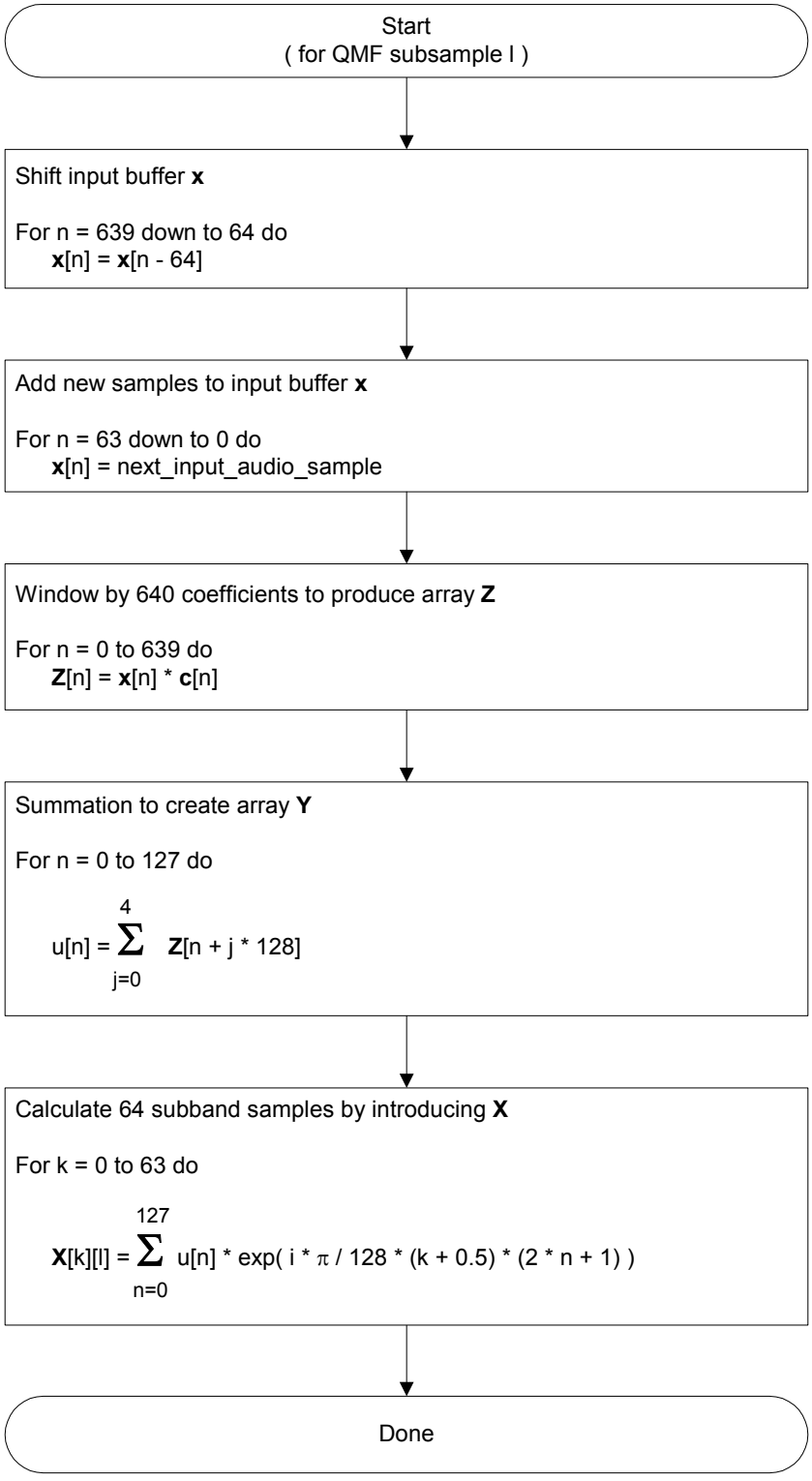


Figure 4.B.12 – Flowchart of encoder analysis QMF bank

4.B.18.3 Time / frequency grid generation

On the input signal, analysis is performed. Information obtained from this analysis is used to choose the appropriate time/frequency resolution of the current SBR frame. The algorithm calculates the start and stop time borders of the SBR envelopes in the current SBR frame, the number of SBR envelopes, as well as their frequency resolution. The different frequency resolutions are calculated as described in subclause 4.6.18.3. The algorithm also calculates the number of noise floors for the given SBR frame and start and stop time borders of the same. The start and stop time borders of the noise floors should be a subset of the start and stop time borders of the spectral envelopes. The algorithm divides the current SBR frame into four classes:

FIXFIX - Both leading and trailing time borders equal nominal frame boundaries. All SBR envelope time borders in the frame are uniformly distributed in time.

FIXVAR - Leading time border equals leading nominal frame boundary. Trailing time border uses time border according to data element *bs_var_bord_1*. All SBR envelope time borders between the leading and trailing time border are specified by **bs_rel_bord_1** as the relative distance in time slots to previous border, starting from the trailing time border.

VARFIX - Leading time border uses time border according to data element *bs_var_bord_0*. Trailing time border equals trailing nominal frame boundary. All SBR envelope time borders between the leading and trailing time border are specified by **bs_rel_bord_0** as the relative distance in time slots to previous border, starting from the leading time border.

VARVAR - Leading time border uses time border according to data element *bs_var_bord_0*. Trailing time border uses time border according to data element *bs_var_bord_1*. All SBR envelope time borders between the leading and trailing time borders are specified by **bs_rel_bord_0** and **bs_rel_bord_1**. The relative time borders starting from the leading time border are specified by **bs_rel_bord_0** as the relative distance to previous time border. The relative time borders starting from the trailing time border are specified by **bs_rel_bord_1** as the relative distance to previous time border.

There are no restrictions on SBR frame class transitions, i.e. any sequence of classes is allowed. However, the values of *bs_var_bord_** must be selected such that the leading border (SBR frame boundary) of the current SBR frame coincides with the trailing border (SBR frame boundary) of the previous SBR frame. Of course the values of *bs_num_rel_** and *bs_rel_bord_** must be selected such that all corresponding borders fall within the boundaries of the SBR frame in question. Furthermore, the maximum number of SBR envelopes per SBR frame is restricted to 4 for class FIXFIX (*bs_num_env* = {1,2,4}) and 5 for class VARVAR (using arbitrary combinations of values of *bs_num_rel_0* and *bs_num_rel_1*). Classes FIXVAR and VARFIX are syntactically limited to 4 SBR envelopes.

4.B.18.4 Envelope estimation

The spectral envelopes of the current SBR frame are estimated over the time segment and with the frequency resolution given by the time/frequency grid represented by $\mathbf{t}_E$ and $\mathbf{r}$. The SBR envelope is estimated by averaging the squared complex subband samples over the given time/frequency regions.

$$\mathbf{E}(k-k_x,l)=\frac{\sum_{i=RATE\cdot\mathbf{t}_E(l)}^{RATE\cdot\mathbf{t}_E(l+1)-1}\sum_{j=k_l}^{k_h-1}\left|\mathbf{X}(j,i)\right|^2}{\left(RATE\cdot\mathbf{t}_E(l+1)-RATE\cdot\mathbf{t}_E(l)\right)\cdot(k_h-k_l)},\quad k_l\le k<k_h,\begin{cases}k_l=\mathbf{F}(p,\mathbf{r}(l))\\k_h=\mathbf{F}(p+1,\mathbf{r}(l))\end{cases},0\le p<\mathbf{n}(\mathbf{r}(l)),0\le l<L_E$$

In the case of stereo and coupling the energy is calculated according to:

$$\mathbf{E}_{Left}(k-k_x,l)=\frac{\sum_{i=RATE\cdot\mathbf{t}_E(l)}^{RATE\cdot\mathbf{t}_E(l+1)-1}\sum_{j=k_l}^{k_h-1}\left(\left|\mathbf{X}_{Left}(j,i)\right|^2+\left|\mathbf{X}_{Right}(j,i)\right|^2\right)}{2\cdot\left(RATE\cdot\mathbf{t}_E(l+1)-RATE\cdot\mathbf{t}_E(l)\right)\cdot(k_h-k_l)}$$

$$\mathbf{E}_{Right}\left(k-k_x,l\right)=\frac{\varepsilon+\sum_{i=RATE\cdot\mathbf{t}_E(l)}^{RATE\cdot\mathbf{t}_E(l+1)-1}\sum_{j=k_l}^{k_h-1}\left|\mathbf{X}_{Left}\left(j,i\right)\right|^2}{\varepsilon+\sum_{i=RATE\cdot\mathbf{t}_E(l)}^{RATE\cdot\mathbf{t}_E(l+1)-1}\sum_{j=k_l}^{k_h-1}\left|\mathbf{X}_{Right}\left(j,i\right)\right|^2}$$

for $k_l \le k < k_h, \begin{cases} k_l = \mathbf{F}\left(p,\mathbf{r}\left(l\right)\right) \\ k_h = \mathbf{F}\left(p+1,\mathbf{r}\left(l\right)\right) \end{cases}, 0 \le p < \mathbf{n}\left(\mathbf{r}\left(l\right)\right), 0 \le l < L_E$.

For stereo with no channel coupling, the energy for every channel is calculated as in the mono case.

4.B.18.5 Additional control parameters

In order to achieve optimal results, given the HF generator used in the decoder, several additional parameters apart from the spectral envelope are assessed. The noise floor scalefactor is estimated for the current SBR frame. It is defined as the ratio between the energy of the noise that should be added to a particular frequency band, in order to obtain a similar tonal to noise ratio to that of the original signal, and the energy of the HF generated signal for that frequency band.

$$\mathbf{Q}=\frac{Energy_{Additional\_Noise}}{Energy_{HF\_Generated}}$$

The noise floor scalefactor is estimated once or twice per SBR frame dependent on the number of spectral envelopes estimated for the SBR frame (indicated by $\mathbf{t}_Q$). The frequency resolution for the noise floor scalefactor is calculated according to the same algorithm subsequently used in the decoder and described in the subclause 4.6.18.3. The start and stop time borders of the different noise floors are given from the time grid.

The level of the inverse filtering applied in the decoder is estimated for different frequency ranges with the same frequency resolution as used for the noise floor scalefactor estimation. The estimation algorithm compares the tonality of the original and the tonality that will be attained after the HF generator in the decoder. The ratio between the two is mapped to four different inverse filtering levels, off, low, mid and high. These levels corresponds to different chirp factors in the HF generator as outlined in subclause 4.6.18.5. Moreover, the encoder assesses where a strong tonal component will be missing after the HF generation in the decoder. This detection is done on the highest frequency resolution given by the high frequency resolution table, $\mathbf{f}_{TableHigh}$. The level of the tonal component is implicitly coded by the SBR envelope and the noise floor scalefactors, and thus only the frequency needs to be coded.

4.B.18.6 Data quantization

The spectral envelope scalefactors are quantized in 3dB steps or in 1.5dB steps, dependent on the time frequency resolution of the current SBR frame, and *bs_amp_res*. For the case where there is only one SBR envelope per SBR frame and of SBR frame class FIXFIX, 1.5 dB steps are always used, disregarded the value of *bs_amp_res*.

For mono and stereo without channel coupling the quantization is done according to:

$$\mathbf{E}_Q\left(k,l\right)=INT\left(a\cdot\max\left(\log_2\left(\frac{\mathbf{E}\left(k,l\right)}{64}\right),0\right)+0.5\right), 0\le k<\mathbf{n}\left(\mathbf{r}\left(l\right)\right), 0\le l<L_E$$

where $a=\begin{cases}2 & ,bs\_amp\_res=0\\ 1 & ,bs\_amp\_res=1\end{cases}$

For the coupled channel mode, the left channel is quantized according to the above, while the right channel should be quantized according to:

$$\mathbf{E}_{QRight}\left(k,l\right)=INT\left(a\cdot\log_2\left(\mathbf{E}\left(k,l\right)\right)+0.5\right)+\mathbf{panOffset}\left(bs\_amp\_res\right)$$

The noise floor scalefactors data is always quantized in 3dB steps according to:

$$\mathbf{Q}_Q(k,l) = INT\left(NOISE\_FLOOR\_OFFSET - \log_2\left(\mathbf{Q}(k,l)\right) + 0.5\right),$$

where $\mathbf{Q}_Q(k,l)$ shall be limited to the interval $[0,30]$.

For stereo the left and right channels are quantized according to the above. For coupling however, the right channel is quantized according to:

$$\mathbf{Q}_{QRight}(k,l) = INT\left(\log_2\left(\frac{\mathbf{Q}_{Left}(k,l)}{\mathbf{Q}_{Right}(k,l)}\right) + 0.5\right) + \mathbf{panOffset}(1),$$

where

$\mathbf{Q}_{QRight}(k,l)$ shall be limited to the interval $\left[0, 2\cdot\mathbf{panOffset}(1)\right]$.

and

$$\mathbf{Q}_{QLeft}(k,l) = INT\left(NOISE\_FLOOR\_OFFSET - \log_2\left(\frac{\mathbf{Q}_{Left}(k,l) + \mathbf{Q}_{Right}(k,l)}{2}\right) + 0.5\right)$$

which is limited to the interval $[0,30]$.

In the case of coupling, the $\mathbf{Q}_{QRight}(k,l)$ and $\mathbf{E}_{QRight}(k,l)$ values shall be quantised to multiples of two, e.g. $[0,2,4,6,8...]$.

4.B.18.7 Envelope coding

The spectral envelope scalefactors are delta coded in either the time direction or the frequency direction, according to the preferred choice indicated in **bs_df_env***(l)* below. The **bs_df_env** elements may be chosen arbitrarily, with the reservation for the case when *reset* = 1. In this case delta coding in the time direction is not allowed for the first SBR envelope of that SBR frame.

$$\mathbf{E}_{Delta}(k,l) = \begin{cases} \delta \cdot \mathbf{E}_Q(0,l) & , \begin{cases} 0 \le l < L_E \\ k = 0 \\ \mathbf{bs\_df\_env}(l) = 0 \end{cases} \\ \delta \cdot \left(\mathbf{E}_Q(k,l) - \mathbf{E}_Q(k-1,l)\right) & , \begin{cases} 0 \le l < L_E \\ 1 \le k < \mathbf{n}(\mathbf{r}(l)) \\ \mathbf{bs\_df\_env}(l) = 0 \end{cases} \\ \delta \cdot \left(g_E(k,l) - \mathbf{E}_Q(k,l)\right) & , \begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \\ \mathbf{bs\_df\_env}(l) = 1 \\ \mathbf{r}(l) = g(l) \end{cases} \\ \delta \cdot \left(g_E(i(k),l) - \mathbf{E}_Q(k,l)\right) & , \begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \\ \mathbf{bs\_df\_env}(l) = 1 \\ \mathbf{r}(l) = 0 \\ g(l) = 1 \\ i(k) \text{ is defined by} \\ \mathbf{f}_{TableHigh}(i(k)) = \mathbf{f}_{TableLow}(k) \end{cases} \\ \delta \cdot \left(g_E(i(k),l) - \mathbf{E}_Q(k,l)\right) & , \begin{cases} 0 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \\ \mathbf{bs\_df\_env}(l) = 1 \\ \mathbf{r}(l) = 1 \\ g(l) = 0 \\ i(k) \text{ is defined by} \\ \mathbf{f}_{TableLow}(i(k)) \le \mathbf{f}_{TableHigh}(k) < \mathbf{f}_{TableLow}(i(k)+1) \end{cases} \end{cases}$$

where $\delta = \begin{cases} 0.5 & if\ ch = 1\ AND\ bs\_coupling = 1 \\ 1 & otherwise \end{cases}$ and,

where $g_E(k,l)$ and $g(l)$ is defined below. As $\mathbf{E}_Q$ represents the envelope scalefactors for the current SBR frame, the envelope scalefactors from the previous SBR frame is denoted $\mathbf{E}'_Q$. Envelope scalefactors from the previous SBR frame, $\mathbf{E}'_Q$ is needed when delta coding in time direction over SBR frame boundaries. The number of SBR envelopes of the previous SBR frame is denoted L'_E, and is also needed in that case, as well as frequency resolution vector of the previous SBR frame, denoted **r'**.

$$g_E(k,l) = \begin{cases} \mathbf{E}_Q(k,l-1) & , \begin{cases} 1 \le l < L_E \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \end{cases} \\ \mathbf{E}'_Q(k, L'_E - 1) & , \begin{cases} l = 0 \\ 0 \le k < \mathbf{n}(\mathbf{r}(l)) \end{cases} \end{cases} \text{ and } g(l) = \begin{cases} \mathbf{r}(l-1) & , 1 \le l < L_E \\ \mathbf{r}'(L'_E - 1) & , l = 0 \end{cases}.$$

The same is true for the noise floor scalefactors.

For the envelope scalefactors and the noise floor scalefactors different Huffman tables are used dependent on coding directions, quantization and stereo mode, according to the Table 4.A.76 in subclause 4.A.6.1.

Contents for Subpart 5

Subpart 5: Structured Audio (SA)

5.1 Scope

5.1.1 Overview of subpart

5.1.1.1 Purpose

The Structured Audio toolset enables the transmission and decoding of synthetic sound effects and music by standardising several different components. Using Structured Audio, high-quality sound can be created at extremely low bandwidth. Typical synthetic music may be coded in this format at bitrates ranging from 0 kbps (no continuous cost) to 2 or 3 kbps for extremely subtle coding of expressive performance using multiple instruments.

MPEG-4 does not standardise a particular set of synthesis methods, but a method for describing synthesis methods. Any current or future sound-synthesis method may be described in the MPEG-4 Structured Audio format.

5.1.1.2 Introduction to major elements

There are five major elements to the Structured Audio toolset:

1. The Structured Audio Orchestra Language, or SAOL. SAOL is a digital-signal processing language which allows for the description of arbitrary synthesis and control algorithms as part of the content bitstream. The syntax and semantics of SAOL are standardised here in a normative fashion.

2. The Structured Audio Score Language, or SASL. SASL is a simple score and control language which is used in certain object types (see subclause 5.6) to describe the manner in which sound-generation algorithms described in SAOL are used to produce sound.

3. The Structured Audio Sample Bank Format, or SASBF. The Sample Bank format allows for the transmission of banks of audio samples to be used in wavetable synthesis and the description of simple processing algorithms to use with them.

4. A normative scheduler description. The scheduler is the supervisory run-time element of the Structured Audio decoding process. It maps structural sound control, specified in SASL or MIDI, to real-time events dispatched using the normative sound-generation algorithms.

5. Normative reference to the MIDI standards, standardised externally by the MIDI Manufacturers Association. MIDI is an alternate means of structural control which can be used in conjunction with or instead of SASL. Although less powerful and flexible than SASL, MIDI support in this standard provides important backward-compatibility with existing content and authoring tools. MIDI support in this standard consists of a list of recognised MIDI messages and normative semantics for each.

5.2 Normative references

[DLS] (c) 1997 MIDI Manufacturers Association, *The MIDI Downloadable Sounds Specification, v. 97.1.*

[DLS2] (c) 1998 MIDI Manufacturers Association, *The MIDI Downloadable Sounds Specification, v. 98.2*.

[MIDI] (c) 1996 MIDI Manufacturers Association, *The Complete MIDI 1.0 Detailed Specification v. 96.2.*

5.3 Definitions

Definitions can be found in subpart 1, subclause 1.3.

5.4 Symbols and abbreviations

5.4.1 Mathematical operations

The mathematical operators used to describe this part of ISO/IEC 14496 are similar to those used in the C programming language.

+	addition
-	subtraction
x or *	multiplication
/	division
exp	exponential function (base *e*)
log	natural logarithm
log10	base-10 logarithm
abs	absolute value
floor(x)	greatest integer less than or equal to x
ceil(x)	least integer greater than or equal to x
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<> or !=	not equal to

5.4.2 Description methods

5.4.2.1 Bitstream syntax

The Structured Audio bitstream syntax is described using SDL, the MPEG-4 Syntactic Description Language. See ISO/IEC 14496-1 Subpart 12.

5.4.2.2 SAOL syntax

The textual SAOL syntax (in subclause 5.8) is described using extended Backus-Naur format (BNF) notation [see **DRAG**]. BNF is a description for context-free grammars of programming languages.

BNF grammars are composed of terminals, also called tokens, and production rules. Terminals represent syntactic elements of the language, such as keywords and punctuation; production rules describe the composition of these elements into structural groups.

Terminals will be represented in **boldface**; production rules will be represented in <angle brackets>.

The rewrite rules, which map productions into sequences of other productions and terminals, are represented with the -> symbol.

EXAMPLE

<letter> -> **a**
<letter> -> **b**
<sequence> -> <letter>
<sequence> -> <letter> <sequence>

This grammar (starting from the *sequence* token) describes, using a recursive rewrite rule and a two-symbol alphabet, all strings containing at least one letter that are made up of ‘a’ and ‘b’ characters.

In addition, rewrite rules using optional elements will be described using the [ ] symbols. Using this notation does not increase the power of the syntax description (in terms of the languages it can represent), but makes certain constructs simpler.

EXAMPLE

<head> -> **c**
<seqhead> -> [<head>] <sequence>

This grammar (starting from the seqhead token) describes, in addition to the set above, all strings beginning with a ‘c’ character and followed by a sequence of ‘a’s and ‘b’s.

The **NULL** token may be used to indicate that a sequence of no characters (the empty string) is a permissible rewrite for a particular production.

Other symbols such as the ellipsis (...) will be used occasionally when their meaning is clear from the context.

Normative aspects of the relationship between the BNF grammar, other grammar representation methods, the bitstream syntax, and the textual description format are described in subclause 5.8.1.

5.4.2.3 SASL Syntax

The SASL syntax is specified using extended BNF grammars, as described in subclause 5.4.2.2.

5.5 Bitstream syntax and semantics

5.5.1 Introduction to bitstream syntax

This subclause describes the bitstream format defining an MPEG-4 Structured Audio bitstream.

Each group of classes is notated with normative semantics, which define the meaning of the data represented by those classes.

5.5.2 Bitstream syntax

```
/*********************************
      symbol table definitions
*********************************/

class symbol {
  unsigned int(16) sym;         // no more than 65535 symbols/orch + score
}

class sym_name {                // one name in a symbol table
  unsigned int(4) length;       // names up to 15 chars long
  unsigned int(8) name[length];
}

class symtable {                // a whole symbol table
  unsigned int(16) length;      // no more than 65535 symbols/orch+score
  sym_name name[length];
}
```

A bitstream may contain a symbol table, but this is not required. The symbol table allows textual SAOL and SASL code to be recovered from the tokenised bitstream representation. The inclusion or exclusion of a symbol table does not affect the decoding process.

If a symbol table is included, then all or some of the symbols in the orchestra and score shall be associated with a textual name in the following way: each symbol (a symbol is just an integer) shall be associated with the character string paired with that symbol in a `sym_name` object. There shall be no more than one name associated with a given symbol, otherwise the bitstream is invalid. It is permissible for the symbol table to be incomplete and contain names associated with some, but not all, symbols used in the orchestra and score. The presence of a zero-length string in a symbol table entry indicates that a name for this symbol is not included in the symbol table

SAOL and SASL implementations that require textual input, rather than tokenised input, are permissible in a compliant decoder, in which case the decoder would detokenise the bitstream before it can be processed. In such a case, any symbols without associated names are suggested to be associated with a default name of the form `_sym_x`, where x is the symbol value. Names of this form are reserved in SAOL for this purpose, and so following this suggestion guarantees that names will not clash with symbol-table-defined symbol names.

```
/*********************************
    orchestra file definitions
*********************************/

class orch_token {              // a token in an orchestra
  int done;

  unsigned int(8) token;         // see standard token table, Annex 5.A
  switch (token) {
  case 0xF0 :                   // a symbol
    symbol sym;                 // the symbol name
    break;
  case 0xF1 :                   // a constant value
    float(32) val;              // the floating-point value
    break;
  case 0xF2 :                    // a constant int value
    unsigned int(32) val;       // the integer value
    break;
  case 0xF3 :                   // a string constant
    int(8) length;
    unsigned int(8) str[length]; // strings no more than 255 chars
    break;
  case 0xF4 :                   // a one-byte constant
    int(8) val;
    break;
  case 0xFF : // end of orch
    done = 1;
    break;
  }
}

class orc_file {                // a whole orch file
  unsigned int(16) length;
  orch_token data[length];
}
```

An orchestra file is a string of tokens. These tokens represent syntactic elements such as reserved words, core opcode names, and punctuation marks as given in the table in Annex 5.A; in addition, there are five special tokens. Token 0xF0 is the symbol token; when it is encountered, the next 16 bits in the bitstream shall be a symbol number. Token 0xF1 is the value token; when it is encountered, the next 32 bits in the bitstream shall be a floating-point value. This token shall be used for all symbolic constants within the SAOL program except for those encountered in special integer contexts, as described in subclause 5.12. Token 0xF2 is the integer token; when it is encountered, the next 32 bits in the bitstream shall be an unsigned integer value. Token 0xF3 is the string token; when it is encountered, the next several bits in the bitstream shall represent a character string (this token is currently unused). Token 0xF4 is the byte token; when it is encountered, the next 8 bits in the bitstream shall be an unsigned integer value. Token 0xFF is the end-of-orchestra token; this token has no syntactic function in the SAOL orchestra, but signifies the end of the orchestra file section of the bitstream.

Not every sequence of tokens is permitted to occur as an orchestra file. Subclause 5.8 contains extensive syntactic rules restricting the possible sequence of tokens, described according to the textual SAOL format. Normative rules for mapping back and forth between the tokenised format and the textual format are given in subclause 5.12. The overall sequence of orchestra tokens shall correspond to an <orchestra> production as given in subclause 5.8.4.

```
/*********************************
      score file definitions
*********************************/

class instr_event {            // a note-on event
  bit(1) has_label;
  if (has_label)
```

```
    symbol label;
  symbol iname_sym;            // the instrument name
  float(32) dur;               // note duration
  unsigned int(8) num_pf;
  float(32) pf[num_pf];        // all the pfields (no more than 255)
}

class control_event {          // a control event
  bit(1) has_label;
  if (has_label)
    symbol label;
  symbol varsym;               // the controller name
  float(32) value;             // the new value
}

class table_event {
  symbol tname;       // the name of the table
  bit(1) destroy;     // a table destructor
  if (!destroy) {
    token tgen;         // a core wavetable generator
    bit(1) refers_to_sample;
    if (refers_to_sample)
      symbol table_sym;          // the name of the sample
    unsigned int(16) num_pf;     // the number of pfields
    if (tgen == 0x7D) {           // concat
       float(32) size;
       symbol ft[num_pf - 1];
    } else {
       float (32) pf[num_pf];
    }
    // when coding sample generator, leave a blank array slot
    // for "which" parameter, to maintain alignment for "skip" parameter
  }
}

class end_event {
  // fixed at nothing
}

class tempo_event {  // a tempo event
  float(32) tempo;
}

class score_line {
  bit(1) has_time;
  if (has_time) {
     bit (1) use_if_late;
     float(32) time;              // the event time
  }
  bit (1) high_priority;
  bit(3) type;
  switch (type) {
    case 0b000 : instr_event inst; break;
    case 0b001 : control_event control; break;
    case 0b010 : table_event table; break;
    case 0b100 : end_event end; break;
    case 0b101 : tempo_event tempo; break;
  }
}

class score_file {
  unsigned int(20) num_lines;  // a whole score file
  score_line lines[num_lines];
}
```

A score file is a set of lines of score information provided in the stream information header. Thus, events that are known before the real-time bitstream transmission begins may be included in the header, so that they are available to the decoder immediately, which may aid efficient computation in certain implementations. Each line shall be one of five events. Each type of event has different implications in the decoding and scheduling process, see subclause 5.7.3. An instrument event specifies the start time, instrument name symbol, duration, and any other parameters of a note played on a SAOL instrument. A control event specifies a control parameter that is passed

to a instrument or instruments already generating sound. A table event dynamically creates or destroys a global wavetable in the orchestra. An end event signifies the end of orchestra processing. A tempo event dynamically changes the tempo of orchestra playback.

A score file need not be presented in increasing order of event times; the events shall be "sorted" by the scheduler as they are processed. In the score file, every score line shall have a time stamp (**has_time** shall be 1).

The **high_priority** bit indicates that the score event is a high-priority event as described in subclause 5.7.3.3.7. The **use_if_late** bit indicates, if the **has_time** bit is set, that the score event shall be used whether or not it arrives on time (see subclause 5.7.3.3.8).

```
/*********************************
         MIDI definitions
*********************************/

class midi_event {
  unsigned int(24) length
  unsigned int(8) data[length];
}

class midi_file {
  unsigned int(32) length;
  unsigned int(8) data[length];
}
```

The MIDI chunks allow the inclusion of MIDI score information in the bitstream header and bitstream. The MIDI event class contains a single MIDI instruction as specified in **[MIDI]**; the MIDI file class contains an array of bytes corresponding to a Standard Format 0 or Format 1 MIDIFile as specified in **[MIDI]**. Note that not every sequence of data may occur in either case; the legal syntaxes of MIDI events and MIDIFiles as specified in **[MIDI]** place normative bounds on syntactically valid MPEG-4 Structured Audio bitstreams. Only chunks of data up to 2^{24}-1 and 2^{32}-1 bytes long, respectively, may be included; longer messages shall be broken into several bitstream elements. The semantics of MIDI data are given in subclause 5.13 (for Object type 1 and 2 implementations) and subclause 5.14 (for Object type 3 and 4 implementations).

```
/*********************************
         sample data
*********************************/

class sample {
  /* note that 'sample' can be used for any big chunk of data
     that needs to get into a wavetable */
  symbol sample_name_sym;
  unsigned int(24) length;  // length in samples
  bit(1) has_srate;
  if (has_srate)
    unsigned int(17) srate; // sampling rate (needs to go to 96 KHz)
  bit(1) has_loop;
  if (has_loop) {
    unsigned int(24) loopstart; // loop points in samples
    unsigned int(24) loopend;
  }
  bit(1) has_base;
  if (has_base)
    float(32) basecps;         // base freq in Hz
  bit(1) float_sample;
  if (float_sample) {
    float(32) float_sample_data[length]; // data as floats ...
  }
  else {
    int(16) sample_data[length]; // ... or as ints
  }
}
```

A sample chunk includes a block of data that will be included in a wavetable in a SAOL orchestra. Each sample consists of a name, a length, a block of data, and four optional parameters: the sampling rate, the loop start and

loop end points, and the base frequency. Access to the data in the sample is provided through the **sample** core wavetable generator, see subclause 5.10.2.

The sample data may be represented either as 32-bit floating point values, in which case it shall be scaled between –1 and 1, or may be represented as 16-bit integer values, in which case it shall be scaled between -32768 and 32767. In the case that the sample data is represented as integer values, upon inclusion in a wavetable, it shall be rescaled to floating-point as described in subclause 5.10.2.

Each sample is named with a symbol. If two samples in the decoder configuration header or in a single access unit have the same name, the result is unspecified. If a sample in an access unit has the same name as a sample in a previous access unit or one in the decoder configuration header, the new sample shall replace the old sample for accesses to that name through the **sample** core wavetable generator for any table generator executed at the same time or later than the decoding time of the access unit containing the new sample. Tables that have already been generated are not affected.

```
/************************
       sample bank data
************************/
```

The sample bank chunk describes a bank of wavetable data and associated processing parameters for use with the sample bank synthesis procedure in subclause 5.13.

```
class sbf {
       int(32)             length;
       int(8)              data[length];
}
```

The data chunk is opaque with regard to transport by MPEG-4 Systems. It shall conform to the format specification given in [**DLS**] (see subclause 5.2) – that is, it shall be a RIFF data chunk beginning “RIFF...”.

```
/*********************************
         bitstream formats
*********************************/

class StructuredAudioSpecificConfig {  // the bitstream header
  bit more_data = 1;

  while (more_data) {  // shall have at least one chunk
    bit(3) chunk_type;
    switch (chunk_type) {
    case 0b000 : orc_file orc; break;
    case 0b001 : score_file score; break;
    case 0b010 : midi_file SMF; break;
    case 0b011 : sample samp; break;
    case 0b100 : sbf sample_bank; break;
    case 0b101 : symtable sym; break;
    }
    bit(1) more_data;
  }
}
```

The bitstream decoder configuration contains all the information required to configure and start up a structured audio decoder. It contains a sequence of one or more chunks, where each chunk is of one of the following types: orchestra file, score file, midi file, sample data, sample bank, or symbol table. Multiple chunks of each of these types may occur in the bitstream (except for **midi_file**), with the following semantics:

1. **orc_file**: The multiple orchestra files shall be merged. It is a syntax error if more than one **global** block appears in the merged orchestra (see subclause 5.8.5).

2. **score_file**: The multiple score files shall be sorted together by event times and merged.

3. **midi_file**: Only one **midi_file** element may occur in a single Structured Audio bitstream.

4. **sample**: The samples may be accessed by the orchestra as described in subclause 5.10.2.

5. **sbf**: The multiple sample banks are all accessible to the synthesis process. The behaviour is undefined if a particular combination of MIDI present and MIDI bank number is used more than once, whether in a single sample bank or in multiple sample banks.

6. **symtable**: The multiple symbol tables each give names to symbols in the orchestra. The N_0 names in the first symbol table in the bitstream apply to symbols 0..N_0-1; the N_1 names in the second symbol table to symbols N_0..N_0+N_1-1; and so on.

```
class SA_access_unit {      // the streaming data
 bit(1) more_data = 1;

 while (more_data) {
   bit(2) event_type;
   switch (event_type) {
     case 0b00 : score_line score_ev; break;
     case 0b01 : midi_event midi_ev; break;
     case 0b10 : sample samp; break;
     }
   bit(1) more_data;
   }
}
```

The Structured Audio access unit contains real-time streaming control information to be provided to a running Structured Audio decoding process. It may contain as many control instructions as desired and as permitted by the available bandwidth. It shall not contain new instrument definitions; the orchestra configuration is fixed at decoder start-up. It may contain score lines, MIDI events, and new sample data. When provided as part of an access unit, the score line is not required to contain a timestamp. When **has_time** is cleared in the **score_line** class, the event is dispatched immediately according to the rules in subclause 5.7.3.3.6. Score lines without timestamps are not responsive to orchestra tempo changes.

Annex 5.E discusses when the random access point flag, conveyed in the Access Unit packaging in the Systems specification, may be set.

5.6 Object types

There are four object types standardised for Structured Audio. Each of these object types corresponds to a particular set of application requirements. The default object type is Object type 4; when reference is made to MPEG-4 Structured Audio format without reference to a object type, it shall be understood that the reference is to Object type 4.

Terminals implementing MPEG-4 Systems profiles containing the **AudioFX** node (see ISO/IEC 14496-1, subclause 9.4.2.7) shall also provide support for Structured Audio Object type 3 or 4.

1. MIDI only. In this object type, only the **midi_file** chunk shall occur in the stream information header, anzd only the **midi_event** event shall occur in the bitstream data. In this object type, the General MIDI patch mappings are used, and the decoding process is described in subclause 5.14.2. This object type is used to enable backward-compatibility with existing MIDI content and rendering devices. Normative and implementation-independent sound quality cannot be produced in this object type.

2. **Wavetable** synthesis. In this object type, only the **midi_file** and **sbf** chunks shall occur in the stream information header, and only the **midi_event** event shall occur in the bitstream data. This object type is used to describe music and sound-effects content in situations in which the full flexibility and functionality of SAOL, including 3-D audio, is not required. In this case, the decoding process is described in subclause 5.13.3.1.

3. Algorithmic synthesis and AudioFX. In this object type, the **sbf** and **midi_file** chunks shall not occur in the **stream** information header. This object type is used to describe algorithmic synthesis and to provide audio effects processing in the AudioFX node when the use of the SASBF sample bank format (subclause 5.13) is not needed.

4. Main synthetic. All bitstream elements and stream information elements may occur.

The decoding process for Object types 3 and 4 is described in subclause 5.7.

5.7 Decoding process

5.7.1 Introduction

This subclause describes the algorithmic structured audio decoding process, in which a bitstream conforming to Object type 3 or 4 is converted into sound. The decoding process for Object type 1 bitstreams is described in subclause 5.14.2, and the decoding process for Object type 2 bitstreams in subclause 5.13.3.1.

5.7.2 Decoder configuration header

At the creation of a Structured Audio Elementary Stream, a Structured Audio decoder is instantiated and a bitstream object of class **SA_decoder_config** provided to that decoder as configuration information. At this time, the decoder shall initialise a run-time scheduler, and then parse the decoder configuration header into its component parts and use them as follows:

- Orchestra file: The orchestra file shall be checked for syntactic conformance with the SAOL grammar and rate semantics as specified in subclause 5.8. Whatever pre-processing (i.e., compilation, allocation of static storage, etc.) needs to be done to prepare for orchestra run-time execution shall be performed.

- Score file: Each event in the score file shall be registered with the scheduler. To “register” means to inform the scheduler of the presence of a particular parameterised event at a particular future time, and the scheduler’s associated actions.

- MIDI file: Each event in the MIDI file shall be converted into an appropriate event as described in subclause 5.13, and those events registered with the scheduler.

- Sample bank: The data in the bank shall be stored, and whatever pre-processing necessary to prepare for using the bank for synthesis shall be performed.

- Sample data: The data in the sample shall be stored, and whatever pre-processing necessary to prepare the data for reference from a SAOL wavetable generator shall be performed. If the sample data is represented as 16-bit integers in the bitstream, it shall be converted to floating-point format at this time.

- Symbol table: No normative decoder behaviour is associated with the symbol table.

If there is more than one orchestra file in the stream information header, the various files are combined together via concatenation and processed as one large orchestra file. That is, each orchestra file within the bitstream refers to the same global namespace, instrument namespace, and opcode namespace.

5.7.3 Bitstream data and sound creation

5.7.3.1 Relationship with systems layer

At each time step within the systems operation, the systems layer may present the Structured Audio decoder with an Access Unit containing data conforming to the **SA_access_unit** class. The run-time responsibility of the Structured Audio decoder is to receive these AU data elements, to parse and understand them as the various Structured Audio bitstream data elements, to execute the on-going SAOL orchestra to produce one Composition Unit of output, and to present the systems layer with that Composition Unit.

5.7.3.2 Bitstream data elements

As Access Units are received from the systems demultiplexer, they are parsed and used by the Structured Audio decoder in various ways, as follows:

- Sample data shall be stored, and whatever pre-processing is necessary for reference by forthcoming score lines containing references to that sample shall be performed. If the sample data is represented as 16-bit integers in the bitstream, it shall be converted to floating-point format at this time. Any samples in an access unit shall be processed before score lines, in case the score lines reference the samples.

- Score line events shall be registered with the scheduler if they have time stamps, or executed in the next k-cycle, if not.

- MIDI events shall be converted into appropriate SAOL events (see subclause 5.14) and then registered with the scheduler, if they have time stamps, or executed in the next k-cycle, if not.

5.7.3.3 Scheduler semantics

5.7.3.3.1 Purpose of scheduler

The scheduler is the central control mechanism of a Structured Audio decoding system. It is responsible for handling events by instantiating and terminating instruments, keeping track of what instrument instantiations are active, instructing the various instrument instantiations to perform synthesis, routing the output of instruments onto busses, and sending busses to effects instruments. Although there are many ways to perform these tasks, the exact nature of what must be done can be clearly specified. This subclause provides normative bounds on the activities of the scheduler.

5.7.3.3.2 Instrument instantiation

To instantiate an instrument is to create data space for its variables and the data space required for any opcodes called by that instrument. When an instrument is instantiated, the following tasks shall be performed. First, space for any parameter fields shall be allocated and their values set according to the p-fields of the instantiating expression or event. Then, space for any locally declared variables shall be allocated and these variable values set to 0. Then, the current values of any imported i-rate variables shall be copied into the local storage space. Then, locally declared wavetables shall be created and filled with data according to their declaration and the appropriate rules in subclause 5.10.

5.7.3.3.3 Instrument termination

To terminate an instrument instantiation is to destroy the data space for that instance.

5.7.3.3.4 Instrument execution

To execute an instrument instantiation at a particular rate is to calculate the results of the instructions given in that instrument definition. When an instrument instance is executed at a particular rate, the following steps shall be performed. First, the values of any global variables and wavetables imported by that instrument at that rate shall be copied into the storage space of the instrument. In addition, when executing at the a-rate an instrument instance that is the target of a **send** statement, the current value of the **input** standard name in the instance shall be set to the current value of the bus or busses referenced in the **send** statement. Then, the code block for that instrument shall be executed at the particular rate with regard to the data space of the instrument instantiation, as given by the rules in subclause 5.8.6.6. Then, the values of any global variables and wavetables exported by that instrument at that rate shall be copied into the global storage space. Finally, when executing an instrument instantiation at the a-rate, the value of the instance output shall be added to the bus onto which the instrument is routed according to the rules in subclause 5.8.5.4, unless the instance is the target of a **send** expression referencing the special bus **output_bus**, in which case the output of the instrument instance is the output of the orchestra and may be turned into sound.

5.7.3.3.5 Orchestra start-up and configuration

5.7.3.3.5.1 Introduction

This subclause describes the steps required to begin the decoding process. These tasks (determination of instrument and bus width, global variable allocation, **startup** execution, global wavetable creation, bus and **send** instrument initialisation) shall be performed in the order indicated.

5.7.3.3.5.2 Determination of instrument output width and bus width

The output width of each instrument is determined in the order specified by the global sequencing rules (subclause 5.8.5.6); the width of each bus is either provided by a **send** statement (subclause 5.8.5.5) or is determined by the sum of the output widths (subclause 5.8.6.6.8) of the instruments routed to that bus in a single **route** statement (subclause 5.8.5.4). Any instrument that does not receive any bus data according to the sequence rules shall have an **inchannels** width of 1 (this specification is needed since output widths may depend on the value of **inchannels** or the width of **input**).

NOTE: if the output width of an instrument depends on **outchannels**, and the bus width of the bus where that instrument is routed is not specified by means of other **send**/**route** statements, the orchestra may be indeterministic. Programmers should pay special attention in checking the deterministic behavior of the orchestra or make use of determined bus widths (subclause 5.8.5.5).

EXAMPLE 1

Consider the following orchestra.

```
global {
  route(bus1, i1);
  route(bus2, i2, i1);
  send(i2; ; bus1);
  send(i4; ; bus2);
}

instr i1(...) {
  asig a[2];
   ...
  output(a);
}

instr i2(...) {
  asig b;
  ...
  output(input + b);
}


instr i4(...) {
  asig d[inchannels];
  ...
  output(d + input);
}
```

In this orchestra, the global sequencing rules (subclause 5.8.5.6) specify that instrument **i1** precedes instrument **i2,** and instrument **i1** and **i2** precede instrument **i4**. Instrument **i1** has two channels, so bus **bus1** has two channels. Instrument **i2** has two channels, since it gets input from **bus1**. The bus **bus2** has four channels, two each from **i2** and **i1**. The instrument **i4** has four channels, since it gets input from **bus2**.

EXAMPLE 2

Consider the following orchestra.

```
global {
  route(bus1, a);
```

```
  send(b; ;bus1);
  sequence(b,a);
  route(bus2, a, a);
  route(bus2, b);
  send(c; ; bus2);
}

instr a(...) {
  asig x,y;
  output(x,y,x);
}

instr b(...) {
  output(input,input);
}

instr c(...) { ... }
```

This orchestra contains a syntax error. The global sequence rules prescribe that **b** shall precede **a** (since the **sequence** directive overrides the implicit **send** sequencing), and that **a** and **b** precede **c**. The output width of **b** is two channels, since it does not receive any data according to the sequence rules, and so the width of its **input** is one channel. The output width of **a** is three channels. Thus, the width of **bus1** is three channels (although only the first is used by **b** since the width of its **input** is one channel). Thus, the two route statements onto **bus2** are incompatible, since the first uses six channels, but the second only two.

If the **sequence** directive were removed from the global block, then the syntax error would be resolved. In this case, instrument **a** precedes **b**, so **b** has three channels of input and six of output, and **bus2** can be correctly allocated with six channels.

5.7.3.3.5.3 Variable allocation, startup execution and global wavetable creation

Space for any global signal variables (see subclause 5.8.5.3) shall be allocated and their values set to zero. If there is an instrument called **startup** in the orchestra, that instrument shall be instantiated and executed at the i-rate. After this execution is complete, then all global wavetables are created and filled with data according to their definitions in the global block of the orchestra and the appropriate rules in subclause 5.10.

5.7.3.3.5.4 Initialisation of busses and send instruments

After the global wavetable creation, the orchestra busses are created and initialised. Each channel of each bus is set to 0 values. After the busses are created, all instruments that are the targets of **send** statements as described in subclause 5.8.5.5 shall be instantiated and executed at the i-rate in the order specified by the global sequencing rules described in the global block according to subclause 5.8.5.6. Finally, the global absolute orchestra time shall be set to 0.

NOTE - A time is called *absolute* if it is specified in seconds. When a tempo instruction is first decoded and the value of tempo changes from its default value, the score time and the absolute time are not identical anymore; all the times in the score, subsequent to a tempo line execution, are scaled according to the new tempo and enqueued in absolute dispatch and duration times as specified in subclause 5.7.3.3.6, list item 7.

5.7.3.3.6 Decoder execution while streaming

In each orchestra cycle, one Composition Unit of samples is produced by the real-time synthesis process. This synthesis is performed according to the rules below and the resulting orchestra output, as described in list item 11, is presented to the Systems layer as a Composition Unit. To execute one orchestra cycle, the following tasks shall be performed in the order denoted:

1. If there is an end event whose dispatch time is earlier than or equal to the current absolute orchestra time, or an end event has been received without a timestamp since the last execution of this rule, no further output is produced, and all future requests from the systems layer to produce Composition Units are responded to with buffers of all 0s.

2. If there are any instrument events whose dispatch time is earlier than or equal to the current absolute orchestra time, or any instrument events have been received without timestamps since the last execution of this rule, an instrument instantiation is created for each such instrument event (see subclause 5.7.3.3.2), and those instantiations are each executed at the i-rate (see subclause 5.7.3.3.4) in the order prescribed by the global sequencing rules. If the instrument event specifies a duration for that instrument instantiation, the instrument instantiation shall be scheduled for termination at the time given by the sum of the current absolute orchestra time and the specified duration (scaled to absolute time units according to the actual tempo, if any).

   NOTE - If the current orchestra time differs from the instrument dispatch time, the former shall be used to schedule instance termination.

3. If there are any active instrument instantiations whose termination time is earlier than or equal to the current absolute orchestra time, then the **released** standard name (see subclause 5.8.6.8.16) shall be set to 1 within each such instrument instance, and the instance is marked for termination in step 12, below.

4. If there are any control events whose dispatch time is earlier than or equal to the current absolute orchestra time, or any control events have been received without timestamps since the last execution of this rule, the global variables or instrument variables within instrument instantiations labelled by each such control event shall have their values updated accordingly (see subclause 5.11.4). Note that this implies that no more than one control change per variable per control cycle may be received by the orchestra. If multiple control changes that reference the same variable are received in a single control cycle, the resulting value of the instrument or global variable is unspecified.

5. If there are any table events whose dispatch time is earlier than or equal to the current absolute orchestra time, or any table events have been received without timestamps since the last execution of this rule, global wavetables shall be created or destroyed as specified by the table event (see subclause 5.11.5).

6. If there are any MIDI events whose timestamp is earlier than or equal to the current orchestra time, or that have been received without timestamps since the last execution of this rule, they are dispatched according to their semantics in subclause 5.14.3.

7. If there are any tempo events whose dispatch time is earlier than or equal to the current absolute orchestra time, or any tempo events have been received since the last execution of this rule, then the global tempo variable shall be set to the specified value, and the dispatch times of all events pending for execution shall be scaled to new times according to the new tempo value. The already scheduled times for terminations are also scaled in their remaining part, according to the ratio between the old and new tempo. Existing **extend** times are not affected, since they are specified in absolute time and are thus “outside” the score. The value of the **dur** standard name (subclause 5.8.6.8.7) shall be changed in each active instrument instance to reflect the new duration of the instance.

   If multiple tempo events are processed according to the preceding paragraph in the same control cycle, then the global tempo variable shall only be changed once, to the tempo indicated in the tempo event received last or with the latest timestamp, and the other tempo events are discarded. If there are multiple tempo events with the same timestamp, or both an un-timestamped event and a timestamped event shall be dispatched in the same control cycle, then the resulting value of the global tempo variable is unspecified.

   NOTE - If the current orchestra time differs from the tempo dispatch time, the former shall be used to calculate the new durations and future dispatch times of events.

8. If the **speed** field of the **AudioSource** scene node responsible for instantiating this decoder (see subclause 5.15) has been changed in the last k-cycle, the tempo standard variable shall be set to 60 times the value specified in subclause 9.4.2.9 (the **AudioSource** node) of ISO/IEC 14496-1, and events in the orchestra shall be rescaled as specified in list item 7, above.

9. The value of each channel of each bus shall be set to 0.

10. Each active instrument instance shall be executed once at the k-rate and *n* times at the a-rate, where *n* is the number of samples in the control period (see subclause 5.8.5.2.2). Each execution at the k-rate shall be in the order given by the global sequencing rules, and each corresponding execution at the a-rate (that is, the first a-rate execution in a k-cycle of each, the second a-rate execution in a k-cycle of each, etc.) shall be in the order given by the global sequencing rules.

    NOTE 1 - If instrument **a** is sequenced before instrument **b** according to the rules in subclause 5.8.5.6, then the k-rate execution of **a** shall be strictly before the k-rate execution of **b**, and the k-rate execution of **a** shall be strictly before the first a-rate execution of **a**, and the first a-rate execution of **a** shall be strictly before the first a-rate execution of **b**. However, there is no normative sequencing between the second a-rate execution of **a** and the first a-rate execution of **b**, or between the first a-rate execution of **a** and the k-rate execution of **b**, within a particular orchestra cycle.

    NOTE 2 - In accordance with the conformance rules in subclause 5.7.4, the execution ordering described in this subclause may be rearranged or ignored when it can be determined from examination of the orchestra that to do so will have no effect on the output of the decoding process. “Has no effect” shall be taken to mean that the output of the decoding process in rearranged order is sample-by-sample identical to the output of the decoding process performed strictly according to the rules in this subclause.

    NOTE 3 - The k-cycle execution of each instrument instance shall be executed as an atomic operation; that is, the k-cycle execution of one instance shall be completed before the next begins. It is not permissible to execute k-cycles in parallel. This is not true of a-cycles; if two instruments have no sequencing relationship according to the global sequencing rules, their a-cycles may be executed in any order or in parallel.

11. If the special bus **output_bus** is sent to an instrument, the output of that instrument at each a-cycle is the orchestra output at that a-cycle. Otherwise, the value of the special bus **output_bus** after each instrument has been executed for an a-cycle is the orchestra output at that a-cycle. If the value of the current orchestra output is greater than 1 or less than –1, it shall be set to 1 or -1 respectively (hard clipping).

12. For each instance that was marked for termination in step 3, above: if that instrument instance called **extend** with a parameter greater than the amount of time in a control-cycle, the instrument is not terminated. All other instrument instances marked for termination in step 3 are terminated (see subclause 5.7.3.3.3). As discussed in subclause 5.14.3.2.11, in the case of an “All Notes Off” MIDI message, instances may not extend themselves, and are destroyed at this time.

13. The current global absolute orchestra time is advanced by one control period.

5.7.3.3.7 Event priority

Certain events may be specified as “high priority” events, by setting the corresponding field in the bitstream element or using the ***** token in the textual score. In the case that overloading of the capability of the decoder occurs, this flag allows the content author to have a minimum of control on the performance degradation.

If the **high_priority** flag is set, then the event shall always be executed without degradation unless pathological conditions occur and no instantiations created at low priority levels are active. If the **high_priority** flag is cleared, then the event shall be executed without degradation if no critical conditions occur. Instrument events with the **high_priority** flag cleared may be prematurely terminated if resources are not available to dispatch an event with the **high_priority** flag set.

NOTE - Degradation is not intended as an allowed, normative, technique to lower the computational complexity. Conforming decoders shall be able to decode, in normal conditions, bitstreams of the specified Profile@Level with no degradation. Instead, the priority level is intended as a help to implementers in critical situations due to resource overload, for instance in the case of a terminal attempting to decode a more complex bitstream than indicated by the level of the terminal, resource sharing with other applications, or high and unexpected degree of user interaction with the host terminal.

5.7.3.3.8 Late-arriving events

In the case of transmission error or encoder error, certain events may arrive with timestamps that have already passed. The **use_if_late** field in the bitstream element indicates the proper behaviour in this case. If this field is cleared, then the event is ignored, and processing shall continue as if the event had never arrived. If this field is set, then the event is immediately dispatched according to the rules in subclause 5.7.3.3.6 as though it had been received with no timestamp.

5.7.4 Conformance

With regard to all normative language in this subpart of ISO/IEC 14496-3, conformance to the normative language is measured at the time of orchestra output. Any optimisation of SAOL code or rearrangement of processing sequence may be performed as long as to do so has no effect on the output of the orchestra. "Has no effect" in this sense means that the output of the rearranged or optimised orchestra is sample-by-sample identical to the output of the original orchestra according to the decoding rules given in this subpart.

See also ISO/IEC 14496-4.

5.8 SAOL syntax and semantics

5.8.1 Relationship with bitstream syntax

The bitstream syntax description as given in subclause 5.5 specifies the representation of SAOL instruments and algorithms that shall be presented to the decoder in the bitstream. However, the tokenised description as presented there is not adequate to describe the SAOL language syntax and semantics. In addition, for purposes of enabling bitstream creation and exchange in a robust manner, it is useful to have a standard human-readable textual representation of SAOL code in addition to the tokenised binary format.

The Backus-Naur Format (BNF) grammar presented in this subclause denotes a *language*, or an infinite set of *programs*; the legal programs that may be transmitted in the bitstream are restricted to this set. Any program that cannot be parsed by this grammar is not a legal SAOL program – it has a *syntax error* – and a bitstream containing it is an invalid bitstream. Although the bitstream is made up of tokens, the grammar will be described in terms of lexical elements – a *textual representation* – for clarity of presentation. The syntactic rules expressed by the grammar that restrict the set of textual programs also normatively restrict the syntax of the bitstream, through the relationship of the bitstream and the textual format in the normative tokenisation process.

This subclause thus describes a textual representation of SAOL that is standardised, but stands outside of the bitstream-decoder relationship. Subclause 5.12 describes the mapping between this textual representation and the bitstream representation. The exact normative *semantics* of SAOL will be described in reference to the textual representation, but also apply to the tokenised bitstream representation as created via the normative tokenisation mapping.

Annex 5.C contains a grammar for the SAOL textual language, represented in the 'lex' and 'yacc' formats. Using these versions of the grammar, parsers can be automatically created using the 'lex' and 'yacc' tools. However, these versions are for informative purposes only; there is no requirement to use these tools in building a decoder.

Normative language regarding syntax in this subclause provides bounds on syntactically legal SAOL programs, and by extension, the syntactically legal bitstream sequences that can appear in an **orchestra** bitstream class. That is, there are constructions that appear to be permissible upon reading only the BNF grammar, but are disallowed in the normative text accompanying the grammar. The status of such constructions is exactly that of those which are outside of the language defined by the grammar alone. In addition, normative language describing static rate semantics further bounds the set of syntactically legal SAOL programs, and by extension, the set of syntactically legal bitstream sequences.

The decoding process for bitstreams containing syntactically illegal SAOL programs (i.e., SAOL programs that do not conform to the BNF grammar, or contain syntax errors or rate mismatch errors) is unspecified.

Normative language regarding semantics in this subclause describes the semantic bounds on the behaviour of the Structured Audio decoder. Certain constructions describe “run-time error” situations; the behaviour of the decoder in such circumstances is not normative, but implementations are encouraged to recover gracefully from such situations and continue decoding if possible.

5.8.2 Lexical elements

5.8.2.1 Concepts

The textual SAOL orchestra contains punctuation marks, which syntactically disambiguate the orchestra; identifiers, which denote symbols of the orchestra; numbers, which denote constant values; string constants, which are not currently used; comments, which allow internal documentation of the orchestra; and whitespace, which lexically separates the various textual elements. These elements do not occur in the bitstream – since each is represented there by a token – but we define them here to ground the subsequent discussion of SAOL. Within the rest of subclause 5.8, when we discuss the semantics of “an identifier”, this shall be taken to normatively refer to the semantics of the symbol denoted by that identifier; the language used is for clarity of presentation.

A lexical grammar for parsing SAOL, written in the ‘lex’ language, is provided for informative purposes in Annex 5.C.

5.8.2.2 Identifiers

An identifier is a series of one or more letters, digits and the underscore that begins with a letter or underscore; it denotes a symbol of the orchestra. Every identifier that consists of the same characters in the first 16 characters (is equivalent under string comparison to the first 16 characters) denotes the same symbol. Identifiers are case-sensitive, meaning that identifiers that differ only in the case of one or more characters denote different symbols.

A string of characters equivalent to one of the reserved words listed in subclause 5.8.9, to one of the standard names listed in subclause 5.8.6.8, to the name of one of the core opcodes listed in subclause 5.9.3, or to the name of one of the core wavetable generators listed in subclause 5.10 does not denote a symbol, but rather denotes that reserved word, standard name, core opcode, or core wavetable generator.

An identifier is denoted in the BNF grammar below by the terminal symbol **<ident>**.

5.8.2.3 Numbers

There are two kinds of symbolic constants that hold numeric values in SAOL: integer constants and floating-point constants.

The integer constant is required to occur in certain contexts, such as array definitions. An integer token is a series of one or more digits. Since the contexts in which integers are required to occur in SAOL do not allow negative values, there is no provision for negative integers. A string of characters that appears to be a negative integer shall be lexically analysed as a floating-point constant. No integer constant greater than 2^{32} (4294967296) shall occur in the orchestra.

There is no different in SAOL between numbers coded with the bitstream token for integers and those coded with the bitstream token for bytes. The latter is only an aid to compression of the bitstream.

An integer constant is denoted in the BNF grammar below by the terminal symbol **<int>**.

The floating-point constant occurs in SAOL expressions, and denotes a constant numeric value. A floating-point token consists of a base, optionally followed by an exponent. A base is either a series of one or more digits, optionally followed by a decimal point and a series of zero or more digits, or a decimal point followed by a series of one or more digits. An exponent is the letter **e**, optionally followed by either a **+** or **–** character, followed by a series of one or more digits. Since the floating-point constant appears in a SAOL expression, where the unary negation operator is always available, floating-point constants need not be lexically negative. Every floating-point constant in the orchestra shall be representable by a 32-bit floating-point number.

A floating-point constant is denoted in the BNF grammar below by the terminal symbol **<number>**.

5.8.2.4 String constants

String constants are not used in the normative SAOL specification, but a description is provided here so that they may be treated consistently by implementers who choose to add functionality over and above normative requirements to their implementations.

A string constant denotes a constant string value, that is, a character sequence. A string constant is a series of characters enclosed in double quotation marks (**"**). The double quotation character may be included in the string constant by preceding it with a backslash (\) character. Any other character, including the line-break (newline) character, may be explicitly enclosed in the quotation marks.

The interpretation and use of string constants is left open to implementers.

5.8.2.5 Comments

Comments may be used in the textual SAOL representation to internally document an orchestra. However, they are not included in the bitstream, and so are lost on a tokenisation/detokenisation sequence.

A comment is any series of characters beginning with two slashes (**//**), and terminating with a new line. During lexical analysis, whenever the **//** element is found on a line, the rest of the line is ignored.

5.8.2.6 Whitespace

Whitespace serves to lexically separate the various elements of a textual SAOL orchestra. It has no syntactic function in SAOL, and is not represented in the bitstream, so the exact whitespacing of a textual orchestra is lost on a tokenisation/detokenisation sequence. Whitespace is not required in SAOL except so far as to disambiguate tokens and reserved words that appear next to each other (to separate “asig” from the variable name declared, for example).

A whitespace is any series of one or more space, tab, and/or newline characters.

5.8.3 Variables and values

Each signal variable within the SAOL orchestra holds a value, or an ordered set of values for array variables, as an intermediate calculation by the orchestra. At any point in time, the value of a variable, sample in a wavetable, or single element of an array variable, shall be represented by a 32-bit floating-point value.

Conformance to this subclause is in accordance with subclause 5.7.4; that is, implementations are free to use any internal representation for variable values, so long as the results calculated are identical to the results of the calculations using 32-bit floating-point values.

NOTE - For certain sensitive digital-filtering operations, the results of using greater precision in a calculation may be equivalently detrimental to orchestra output as the results of using less precision, as the stability of the filter may be critically dependent on the quantisation error that is provided with 32-bit values. It is strongly discouraged for bitstreams to contain code that generates very different results when calculated with 32-bit and 64-bit arithmetic.

At orchestra output, the values calculated by the orchestra should reside between a minimum value of –1 and a maximum value of 1. These values at orchestra output represent the maximum negatively- and positively-valued audio samples that can be produced by the terminal. If the values calculated by the orchestra fall outside that range, they are clipped to [-1,1] as described in subclause 5.7.3.3 list item 11. This sound is presented, control cycle by control cycle, to the MPEG-4 system for use in AudioBIFS composition.

5.8.4 Orchestra

<orchestra> -> <orchestra element> <orchestra>
<orchestra> -> <orchestra element>

The orchestra is the collection of signal processing routines and declarations that make up a Structured Audio processing description. It shall consist of a list of one or more orchestra elements.

<orchestra element> -> <global block>
<orchestra element> -> <instrument declaration>
<orchestra element> -> <opcode declaration>
<orchestra element> -> <template declaration>
<orchestra element> -> **NULL**

There are four kinds of orchestra elements:

1. The global block contains instructions for global orchestra parameters, bus routings, global variable declarations, and instrument sequencing. It is not permissible to have more than one global block in an orchestra.

2. Instrument declarations describe sequences of processing instructions that can be parametrically controlled using SASL or MIDI score files.

3. Opcode declarations describe sequences of processing instruments that provide encapsulated functionality used by zero or more instruments in the orchestra.

4. Template declarations describe multiple instruments that differ only slightly using a concise parametric form.

Orchestra elements may appear in any order within the orchestra; in particular, opcode definitions may occur either syntactically before or after they are used in instruments or other opcodes.

5.8.5 Global block

5.8.5.1 Syntactic form

<global block> -> **global {** <global list> **}**
<global list> -> <global statement> <global list>
<global list> -> NULL

A global block shall contain a global list, which shall consist of a sequence of zero or more global statements.

<global statement> -> <global parameter>
<global statement> -> <global variable declaration>
<global statement> -> <route statement>
<global statement> -> <send statement>
<global statement> -> <sequence definition>
<global statement> -> <interpolation level>

There are several kinds of global statement.

1. Global parameters set orchestra parameters such as sampling rate, control rate, and number of input and output channels of sound

2. Global variable declarations define global variables that can be shared by multiple instruments.

3. Route statements describe the routing of instrument outputs onto busses.

4. Send statements describe the sending of busses to effects instruments.

5. Sequence definitions describe the sequencing of instruments by the run-time scheduler.

6. The interpolation level specifies the quality of interpolation performed in the synthesis process.

5.8.5.2 Global parameter

5.8.5.2.1 srate parameter

<global parameter> -> **srate <int>;**

The **srate** global parameter specifies the audio sampling rate of the orchestra. The decoding process shall create audio internally at this sampling rate. It is not permissible to simplify orchestra complexity or account for terminal capability by generating audio internally at other sampling rates, for to do so may have seriously detrimental effects on certain processing elements of the orchestra.

The **srate** parameter shall be an integer value between 4000 and 96000 inclusive, specifying the audio sampling rate in Hz. If the **srate** parameter is not provided in an orchestra, the default shall be the fastest of the audio signals provided as input (see subclause 5.15). If the sampling rate is not provided, and there are no input audio signals, the default sampling rate shall be 32000 Hz. It is a syntax error if more than one **srate** parameter instruction occurs in an orchestra.

In a Object type 3 terminal, or when the SAOL orchestra is used in an **AudioFX** AudioBIFS node (see subclause 5.15.3), the **srate** parameter shall only be one of the following values: (4000, 8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000, 88200, 96000).

5.8.5.2.2 krate parameter

<global parameter> -> **krate <int>;**

The **krate** global parameter specifies the control rate of the orchestra. The decoding process shall execute k-rate processing internally at this rate. It is not permissible to simplify orchestra complexity or account for terminal capability by executing k-rate processing at other rates, unless it can be determined that to do so will have no effect on orchestra output. In this case, “no effect” means that the resulting output of the orchestra is sample-by-sample identical to the output created if the control rate is not altered.

The **krate** parameter shall be an integer value between 1 and the sampling rate inclusive, specifying the control rate in Hz. If the **krate** parameter is not provided in an orchestra, the default control rate shall be 100 Hz. It is a syntax error if more than one **krate** parameter instruction occurs in an orchestra.

If the control rate as determined by the previous paragraph is not an even divisor of the sampling rate, then the control rate is the next larger integer that does evenly divide the sampling rate. The *control period* of the orchestra is the number of samples, or amount of time represented by these samples, in one control cycle.

5.8.5.2.3 inchannels parameter

<global parameter> -> **inchannels <int>;**

The **inchannels** global parameter specifies the number of input channels to process. If there are fewer than this many audio channels provided as input sources, the additional channels shall be set to continuous zero-valued signals. If there are more than this many audio channels provided as input sources, the extra channels are ignored.

If the **inchannels** parameter is not provided in an orchestra, the default shall be the sum of the numbers of channels provided by the input sources (see subclause 5.15). If there are no input sources provided, the value shall be 0. It is a syntax error if more than one **inchannels** parameter instruction occurs in an orchestra.

The only normative way in which audio input is processed by a SAOL orchestra is when the orchestra is embedded in an AudioBIFS **AudioFX** node, see ISO/IEC 14496-1 subclause 9.4.2.7. Nonnormative methods for audio processing include soundfile processing or microphone processing, see Annex 5.F.

5.8.5.2.4 outchannels parameter

<global parameter> -> **outchannels <int>;**

The **outchannels** global parameter specifies the number of output channels of sound to produce. The run-time decoding process shall produce and render this number of channels internally. It is not permissible to simplify orchestra complexity or account for terminal capability by producing fewer channels.

If the **outchannels** parameter is not provided in an orchestra, the default shall be one channel. It is a syntax error if more than one **outchannels** parameter instruction occurs in an orchestra.

5.8.5.2.5 interp parameter

<global parameter> -> **interp <int>;**

The **interp** global parameter specifies the quality of interpolation performed in the synthesis process. Various operations require access to wavetables at non-integer points; to access a wavetable at such a point requires interpolation among the available points in the wavetable.

If the **interp** parameter is 0, then "low level" interpolation is performed. Every interpolation shall be performed using a linear interpolation. That is, let **i** and **j** be two consecutive indices of a wavetable, and let **x** and **y** be the values at points **i** and **j** respectively. Assume that the value at point **k** is required, where **i < k < j**. Then let **q** be the value **k** – floor(**k**). Then the interpolated value is **x + q** * (**y – x**).

If the value required is that "between" the last point and first point of the table as it wraps around, consider **x** to be the value of the last point, **y** the value of the first point, and assert **n < k < n+1** where index **n** is the last point in the wavetable. Then calculate the interpolated value as **x** + **q** * (**y – x**) as above.

If the **interp** parameter is 1, then "high level" interpolation is performed. The method of high-level interpolation is non-normative, but it shall be a higher-quality method than linear interpolation.

It is a syntax error if the parameter is not 0 or 1. It is a syntax error if more than one **interp** parameter instruction occurs in an orchestra.

If the **interp** parameter is not specified in an orchestra, then the interpolation quality is "low" by default. If authors wish to have normative, high-quality interpolation for wavetables, they can rewrite their own versions of **tableread**, **oscil**, and other instructions to perform this.

5.8.5.3 Global variable declaration

5.8.5.3.1 Syntactic form

<global variable declaration> -> **ivar** <namelist> **;**
<global variable declaration> -> **ksig** <namelist> **;**
<global variable declaration> -> <table declaration> **;**

Global variable declarations declare variables that may be shared and accessed by all instruments and by a SASL score. Only **ivar** and **ksig** type variables, as well as wavetables, may be declared globally. A global variable declaration is either a table definition, or an allowed type name followed by a list of name declarations.

A global name declaration specifies that a name token shall be created and space equal to one signal value allocated for variable storage in the global context. A global array declaration specifies that a name token shall be created and space equal to the specified number of signal values allocated in the global context.

5.8.5.3.2 Signal variables

<namelist> -> <name>**,** <namelist>
<namelist> -> <name>

A namelist is a sequence of one or more name declarations.

<name> -> **<ident>**
<name> -> **<ident>[**<array length>**]**

<array length> -> **<int>**
<array length> -> **inchannels**
<array length> -> **outchannels**

A name declaration is an identifier (see subclause 5.8.2.2), or an array declaration. For an array declaration, the parameter shall be either an integer strictly greater than 0, or one of the tokens **inchannels** or **outchannels**. If the latter, the array length shall be the same as the number of input channels or output channels of the orchestra, respectively. It is illegal to use the token **inchannels** if the number of input channels to the orchestra is 0.Not every identifier may be used as a variable name; in particular, the reserved words listed in subclause 5.8.8, the standard names listed in subclause 5.8.6.8, the names of the core opcodes listed in subclause 5.9, and the names of the core wavetable generators listed in subclause 5.10 shall not be declared as variable names.

5.8.5.3.3 Wavetable declarations

<table declaration> -> **table <ident> (<ident> ,** <expr> [**,** <expr list>] **) ;**
<expr> as defined in subclause 5.8.6.7.
<expr list> as defined in subclause 5.8.6.6.1.

Wavetables are structures of memory allocated for the typical purpose of allowing rapid oscillation, looping, and playback. The wavetable declaration associates a name (the first identifier) with a wavetable created by a core wavetable generator referenced by the second identifier. It is a syntax error if the second identifier is not one of the core wavetable generators named in subclause 5.10. The first expression in the comma-delimited parameter sequence is termed the *size expression*; the remaining zero or more expressions comprise the *wavetable parameter list*.

The semantics of the size expression and wavetable parameter list are determined by the particular core wavetable generator, see subclause 5.10. Any expression that is i-rate (see subclause 5.8.6.7.2) is legal as part of the table parameter list; in particular, reference to i-rate global variables is allowed (their values may be set by the special instrument **startup**). Each expression shall be single-valued, except in the case of the **concat** generator (subclause 5.10.16), in which case the expressions shall be table references. The order of creation of wavetables is not deterministic, with the exception of the table arguments of a **concat** generator, which are always generated before the **concat** generator that uses them. In this case, the tables used as arguments to the **concat** generator must be appear before the table which uses the **concat** generator, to prevent dependency loops.

It is not recommended for calls to the **tableread()** opcode to occur in the table parameter expressions, and to do so gives unspecified results.

A global wavetable may be referenced by a wavetable placeholder in any instrument or opcode. See subclause 5.8.6.5.4. Global wavetables shall be created and initialised with data at orchestra initialisation time, immediately after the execution of the special instrument **startup.** They shall not be destroyed unless they are explicitly destroyed or replaced by a **table** line in a SASL score.

To create a wavetable, first, the expression fields are evaluated in the order they appear in the syntax according to the rules in subclause 5.8.6.7. Then, the particular wavetable generator named in the second identifier is executed; the normative semantics of each wavetable generator detail exactly how large a wavetable shall be created, and which values placed in the wavetable, for each generator.

5.8.5.4 Route statement

<route statement> -> **route (<ident> ,** <identlist> **) ;**

<identlist> -> **<ident> ,** <identlist>
<identlist> -> **<ident>**
<identlist> -> **<NULL>**

A **route** statement consists of a single identifier, which specifies a bus, and a sequence of one or more instrument names, which specify instruments. The route statement specifies that the instruments listed do not produce sound output directly, but instead their results are placed on the given bus. The output channels from the instruments listed each are placed on a separate channel of the bus. Multiple **route** statements onto the same bus indicate that the given instrument outputs shall be summed on the bus. Multiple **route** statements with differing numbers of channels referencing the same bus are illegal, unless each statement has either *n* channels or 1 channel. In this case, each of the one-channel **route** statements places the same signal on each channel of the bus, which is *n* channels wide.

There shall be at least one instrument name in the instrument list (the NULL subclause in the grammar is provided so that constructions appearing later may use the same production).

EXAMPLES

Assume that instruments **a, b,** and **c** produce one, two, and three channels of output, respectively.

1. The sequence

```
route(bus1, a, b);
route(bus1, c);
```

is legal and specifies a three-channel bus. The first bus channel contains the sum of the output of **a** and the first channel of **c**; the second contains the sum of the first output channel of **b** and the second of **c**; and the third contains the sum of the second channel of **b** and the third channel of **c**.

2. The sequence

```
route(bus1,b);
route(bus1,c);
```

is illegal since the statements refer different numbers of channels to the same bus.

3. The sequence

```
route(bus1,a,c);
route(bus1,a);
route(bus1,b,b);
```

is legal and specifies a four-channel bus. The first and third **route** statements each refer to four channels of audio, and the second refers to one channel, which will be mapped to each of the four channels.

The resulting channel values are as follows, using array notation to indicate the channel outputs from each instrument:

Table 5.1 — Example of calculating bus routing values

Channel	Value
1	`a + a + b[1]`
2	`c[1] + a + b[2]`
3	`c[2] + a + b[1]`
4	`c[3] + a + b[2]`

It is illegal for a **route** statement to reference a bus that is not the special bus **output_bus** and that does not occur in a **send** statement. See subclause 5.8.5.5.

It is illegal for a **route** statement to refer to the special bus **input_bus** (see subclause 5.15.2)**.**

All instruments that are not referred to in **route** statements place their output on the special bus **output_bus,** except for an effect instrument to which **output_bus** was sent (see subclause 5.8.5.5). The same rules for allowable channel combinations to the special bus **output_bus** apply as if the route statements were explicit; these rules are implicit in the rules for the **output** statement, see subclause 5.8.6.6.8.

5.8.5.5 Send statement

<send statement>->**send**(<**ident**>;<expr list>;<namelist>);
<namelist> as defined in subclause 5.8.5.3.2
<expr list> as defined in subclause 5.8.6.6.1

The **send** statement creates an instrument instantiation, defines busses, and specifies that the referenced instrument is used as an effects processor for those busses.

All busses in the orchestra are defined by using **send** statements. It is illegal for a statement referencing a bus to refer to a bus that is not defined in a **send** statement. The exception is the special bus **output_bus**, which is always defined.

The identifier in the **send** statement references an instrument that will be used as a bus-processing instrument, also called *effect instrument*. There is no syntactic distinction between effect instruments and other instruments. The identifier list references one or more busses that shall be made available to the effect instrument through its **input** standard name, as follows:

> The first n_0 channels of **input,** channels 0 through n_0-1 are the n_0 channels of the first referenced bus;
> Channels n_0 through n_0+n_1-1 of **input** are the n_1 channels of the second bus,
> and so forth, with a total of $n_0 + n_1 + \ldots + n_k$ channels.

In addition, the grouping of busses in the **input** array shall be made available to the effect instrument through its **inGroup** standard name, as follows:

> The first n_0 values of **inGroup** have the value 1;
> Channels n_0 through n_0+n_1-1 of **inGroup** have the value 2,
> and so forth, through $n_0 + n_1 + \ldots + n_k$, with the last n_k having the value k.

The expression list is a list of zero or more i-rate expressions that are provided to the effect instrument as its parameter fields. Any expression that is i-rate (see subclause 5.8.6.7.2) is legal as part of this list; in particular, reference to i-rate global variables is allowed. The number of expressions provided shall match the number of parameter fields defined in the instrument declaration; otherwise, it is a syntax error.

The effect instrument referred to in a **send** statement shall be instantiated at orchestra start-up; see subclause 5.7.3.3.5.4. These instrument instantiations shall remain in effect until the orchestra synthesis process terminates. One instrument instantiation shall be created for each **send** statement in the orchestra. If such an instrument instantiation utilises the **turnoff** statement, the instantiation is destroyed (and sound is no longer routed to it). No other changes are made in the orchestra.

Any bus, except for the special bus **output_bus**, may be sent to more than one effect instrument and/or instantiations; in this case, when the simple identifier is not used, it is illegal to refer to a bus with more than one length.The special bus **output_bus** represents the second-to-finalmost processing of a sound stream; it may only be sent to at most one effect instrument, and it is a syntax error if that instrument is itself routed or makes use of the **outbus** statement. If **output_bus** is not sent to an instrument, it is turned into sound at the end of an orchestra cycle (see subclause 5.7.3.3); if **output_bus** is sent to an instrument, the output of that instrument is turned into sound at the end of an orchestra pass. This instrument is not permitted to use the **turnoff** statement.

In the case that the number of input channels received by an instrument instance differs from the width of the bus(es) providing that input (see subclause 5.7.3.3.5.2), the width of **input** and **inGroup** and the value of **inchan** respect the former rather than the latter. In the case that **inchan** is smaller than the number of channels on the buses providing input, only the first **inchan** channels are used; in the case that **inchan** is smaller, the "extra" channels are all 0's in both **input** and **inGroup**.

At least one bus name shall be provided in the **send** instruction.

5.8.5.6 Sequence specification

<sequence specification> -> **sequence (** <identlist> **) ;**
<identlist> as defined in subclause 5.8.5.4.

The **sequence** statement allows the specification of the ordering of execution of instrument instantiations by the run-time scheduler. The identlist references a list of instruments that describes a partial ordering on the set of instruments. If instrument **a** and instrument **b** are referenced in the same **sequence** statement with **a** preceding **b**, then instantiations of instrument **a** shall be executed strictly before instantiations of instrument **b**.

There are several default sequence rules:

1. The special instrument **startup** is instantiated and the instantiation executed at the i-rate at the very beginning of the orchestra.

2. Any instrument instances corresponding to the **startup** instrument are executed first in a particular orchestra cycle.

3. If **output_bus** is sent to an instrument, the instrument instantiation corresponding to that **send** statement is the last instantiation executed in the orchestra cycle.

4. For each instrument routed to a bus that is sent to an effect instrument, instantiations of the routed instrument are executed before instantiations of the effect instrument. If loops are created using **route** and **send** statements, the ordering is resolved syntactically: whichever **send** statement occurs latest, that instrument instantiation is executed latest.

Default rules 2, 3, and 4 may be overridden by use of the **sequence** statement. Rule 1 cannot be overridden.

It is a syntax error if explicit **sequence** statements create loops in ordering. Any **send** statements that are the "backward" part of an implicit **send** loop have no effect.

If the sequence of two instruments is not defined by the default or explicit sequence rules, their instantiations may be executed in any order or in parallel.

It is not possible to specify the ordering of multiple instantiations of the same instrument; these instantiations can be run in any order or in parallel.

EXAMPLES

An orchestra consists of five instruments, **a, b, c, d,** and **e**.

1. The following code fragment

```
route(bus1, a, b);
send(c; ; bus1);
```

is legal and specifies (using the default sequencing rules) that instantiations of instruments **a** and **b** shall be executed strictly before instantiations of instrument **c**. This ordering applies to all instantiations of instrument **c**, not only to the one corresponding to the **send** statement. No ordering is specified between instruments **a** and **b**.

2. The following code fragment

```
route(bus1, a, b);
send(c; ; bus1);
sequence(c,a);
send(d; ; bus1);
```

is legal and specifies that instantiations of instrument **b** shall be executed first, followed by instantiations of instrument **c**, followed by instantiations of instrument **a**, followed by instances of instrument **d**. (See Figure 5.1) The ordering of **b** and **c**, and **a** and **b** with **d**, follows from default rule 3; the placement of instrument **c** follows from the explicit **sequence** statement, which overrides default rule 3. Due to this ordering, the output samples of instrument **a** are not provided to instrument **c** (they get put on the bus "too late"), and however many channels of output this represents are set to 0 in instrument **c**. The output samples of instrument **a** are provided to instrument **d**.

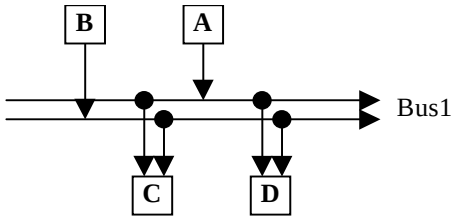


Figure 5.1 — Example of ordering instruments with 'sequence'

3. The following code fragment

```
sequence(a,b);
sequence(b,c,d);
sequence(c,e);
sequence(e,a);
```

is illegal, as it contains an explicit loop in sequencing.

4. The following code fragment

```
route(bus1, a);
send(b; ; bus1);
route(bus2, b);
send(a; ; bus2);
```

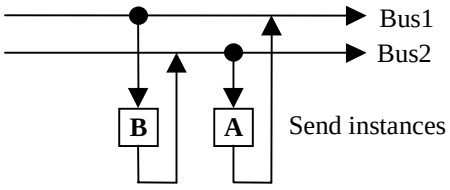


Figure 5.2 — Example of ordering instruments with 'sequence'

is legal, and specifies that instantiations of instrument **b** are executed first, followed by instantiations of instrument **a**. There is an implicit loop here that is resolved syntactically as described in default rule 3. Due to this ordering, the output values of instrument **a** are not provided to instrument **b**. Note that for deciding sequencing, only the order of **send** statements matters, not the order of **route** statements.

5.8.6 Instrument definition

5.8.6.1 Syntactic form

<instrument definition> -> **instr <ident> (** <identlist> **)** [**preset <int>** [**<int> ...**]] **{**
<instr variable declarations>
<block> **}**

An instrument definition has several elements. In order, they are

1. An identifier that defines the name of the instrument,
2. A list of zero or more identifiers that define names for the parameter fields, also called pfields, of the instrument,
3. An optional list of preset values for specifying MIDI preset mappings,
4. A list of zero or more instrument variable declarations, and
5. A block of statements defining the executable functionality of the instrument.

5.8.6.2 Instrument name

Any identifier may serve as the instrument name except that the instrument name shall not be a reserved word (see subclause 5.8.9), the name of a core opcode (see subclause 5.9), or the name of a core wavetable generator (see subclause 5.10). An instrument name may be the same as a variable in local or global scope; there is no ambiguity so created, since the contexts in which instrument names may occur are very restricted.

No two instruments or opcodes in an orchestra shall have the same name.

5.8.6.3 Parameter fields

<identlist> -> as given in subclause 5.8.5.4

The parameter fields, also called pfields, of the instrument, are the interface through which the instrument is instantiated. In the instrument code, the pfields have the rate semantics of i-rate local variables. Their values shall be set on instrument instantiation, before the creation of local variables, with the appropriate values as given in the score line, score event, MIDI event, **send** statement, or **instr** statement corresponding to the instrument instantiation.

5.8.6.4 Preset tag

The **preset** tag specifies the preset number(s) of the instrument. When MIDI program change events arrive in a MIDI stream or MIDI file controlling the orchestra, the program change numbers refer to the **preset** tags given to the various instruments. No more than one instrument may have the same preset number; if multiple instruments in an orchestra specify the same **preset** tag, the one occurring syntactically last is assigned that preset number. If a **preset** tag is not associated with a particular instrument, then that instrument has no preset number and cannot be referenced with a program change. If more than one tag is given, the instrument responds to all of the listed preset values.

Preset tags in SAOL correspond to both the preset and bank value of a program in MIDI control; the program on preset **x**, bank **y** in MIDI syntax shall be indicated as preset (**y** - 1) *128 + (**x** – 1) in SAOL (since presets and banks are numbered starting with 1 in MIDI).

See subclause 5.14 for more normative semantics governing MIDI control of orchestras.

5.8.6.5 Instrument variable declarations

5.8.6.5.1 Syntactic form

<instr variable declarations> -> <instr variable declarations> <instr variable declaration>
<instr variable declarations> -> **<NULL>**

<instr variable declaration> -> [<sharing tag>] **ivar** <namelist> **;**
<instr variable declaration> -> [<sharing tag>] **ksig** <namelist> **;**
<instr variable declaration> -> **asig** <namelist> **;**
<instr variable declaration> -> <table declaration> **;**
<instr variable declaration> -> <sharing tag> **table** <identlist> **;**
<instr variable declaration> -> **oparray <ident> [**<array length> **] ;**
<instr variable declaration> -> <tablemap declaration> **;**

<sharing tag> -> **imports**
<sharing tag> -> **exports**
<sharing tag> -> **imports exports**

<tablemap declaration> -> **tablemap <ident> (** <identlist> **) ;**

<array length> and <namelist> as defined in subclause 5.8.5.3.2, except that in the instrument scope the tokens **inchannels** and **outchannels** refer to the input channels and output channels of the instrument, respectively.
<table declaration> as defined in subclause 5.8.5.3.3
<identlist> as defined in subclause 5.8.5.4

Instrument variable declarations declare variables that may be used within the scope of an instrument. Any rate type variable, as well as wavetables, tablemaps, and wavetable placeholders, may be declared in an instrument. An instrument variable declaration is either a wavetable declaration, or an type name, possibly preceded by a sharing tag, followed by a list of name declarations, or a sharing tag followed by the token **table** followed by a list of identifiers referencing global or future wavetables, or an opcode-array declaration, or a table-map definition.

5.8.6.5.2 Wavetable declaration

The syntax and semantics of subclause 5.8.5.3.3 hold for instrument local wavetables, with the following exceptions and additions:

An instrument local wavetable is available only within the local scope of a single instrument instantiation. As such, it shall be created and initialised with data at the instrument instantiation time, immediately after the pfield values are assigned from the calling parameters. It may be deleted and freed when that instrument instantiation terminates.

Not every expression that is i-rate is legal as part of the table parameter list. Reference to constants, pfields, imported i-rate variables, and i-rate standard names is allowed. However, the instrument wavetable initialisation shall occur before the initialisation pass through the instrument code, and so reference to local i-rate variables is prohibited.

5.8.6.5.3 Signal variables

The syntax and semantics of subclause 5.8.5.3.2 hold for instrument local signal variables, with the following exceptions and additions:

A local name declaration specifies that a name token shall be created and space equal to one signal value allocated for variable storage in each instrument instantiation associated with the instrument definition. A local array declaration specifies that a name token shall be created and space equal to the specified number of signal values allocated in each instrument instantiation associated with the instrument definition.

The sharing tags **imports** and/or **exports** may be used with local i-rate or k-rate signal variable declaration. They shall not be used with a-rate variables. If the **imports** tag is used, then the variable value shall be replaced with

the value of the global variable of the same name at instrument initialisation time (for i-rate signal variables) or at the beginning of each control pass (for k-rate signal variables). The **imports** tag may be used for a local k-rate signal variable even if there is no global variable of the same name, in which case it is an indication that the k-rate variable so tagged may be modified with **control** lines in a SASL score. The **imports** tag shall not be used for local i-rate signal variables when there is no global variable of the same name.

If the **exports** tag is used, then the value of the global variable of the same name shall be replaced with the value of the local signal variable after instrument initialisation (for i-rate signal variables) or at the end of each control pass (for k-rate signal variables). The **exports** tag shall not be used if there is no global variable of the same name.

If, for a particular signal variable, the **imports** and/or **exports** tags are used, and there is a global variable with the same name, then the array width of the local and global variables shall be the same.

If, for a particular local variable, the **imports** tag is not used, then its value is set to 0 before instrument initialisation.

If, for a particular local variable declaration, the **imports** and **exports** tags are not used, even if there is a global variable of the same name, there is no semantic relationship between the two variables. The construction is syntactically legal.

5.8.6.5.4 Wavetable placeholder

The sharing tags **imports** and **exports** may be used to reference global and future wavetables. In this case, the local declaration of the table reference is termed a wavetable placeholder. The wavetable placeholder definition does not contain a full wavetable definition, but only a reference to a global or future wavetable name.

If only the **imports** tag is used, and there is a global wavetable with the same name, then at instrument instantiation time, the current contents of the global wavetable are copied into a local wavetable with that name. If the contents of the global wavetable are modified after a particular instrument instantiation referencing that global wavetable is created, the new contents of the global wavetable shall not be copied into the instrument instantiation. Also, if the contents of the local wavetable are modified, these changes shall not be reflected in the global wavetable.

If the **imports** and **exports** tags are both used, and there is a global wavetable with the same name, then at instrument instantiation time and at the beginning of each control pass, the current contents of the global wavetable are made available to a local wavetable with that name. “Made available” in the preceding sentence means that access may be either in the form of copying data from one wavetable to another or by pointer reference to the same memory space, or by any equivalent implementation. Also, at the end of instrument instantiation and at the end of each control pass, the current contents of the local wavetable are similarly made available to the global wavetable with the same name. In this case, if a wavetable is modified at the a-rate in one instrument instance, it is unspecified exactly when these changes are visible to other instrument instances, but it shall be no later than the next orchestra cycle (it is permitted to be in the same orchestra cycle).

It is not permissible to use the **exports** tag alone for a wavetable placeholder.

If the **imports** tag is used, and there is no global wavetable with the same name, then the reference is to a *future wavetable* that will be provided in the bitstream. When the instrument is instantiated, the contents of the most recent wavetable provided in the bitstream with the same name shall be copied into the local wavetable. If no wavetable has been provided in the bitstream with the same name as the wavetable placeholder at the time of instrument instantiation, then the bitstream is invalid. If the wavetable with this name is changed by providing a new wavetable with the same name by using a **table** line in the bitstream (subclause 5.11.6), the reference immediately changes to the new wavetable when the **table** line is dispatched.

It is not permissible to use the **exports** tag if there is no global wavetable with the same name.

5.8.6.5.5 Opcode array declaration

An opcode array, or “oparray” declaration, declares several opcode states for a particular opcode that may be used by the current instrument or opcode. By declaring the states in this manner, access to them is available through the oparray expression, see subclause 5.8.6.7.7. The identifier in the declaration shall be the name of a core opcode or a user-defined opcode declared elsewhere in the orchestra. The array length declares how many states are available for access to this oparray in the local code block; it shall be an integer value or the special tag **inchannels** or **outchannels.**

It is a syntax error if more than one oparray declaration references the same opcode name in a single instrument or opcode.

5.8.6.5.6 Table map definition

<table map definition> -> **tablemap <ident> (** <identlist> **)**

<identlist> as defined in subclause 5.8.5.4.

A table map is a data structure allowing indirect reference of wavetables via array notation. The identifier names the table map; it shall not be the same as the name of any other signal variable or other restricted word in the local scope. The identifier list gives a number of wavetable names for use with the table map. Each of these names shall correspond to a wavetable definition or wavetable placeholder within the current scope. The **tablemap** declaration may come before, after, or in the midst of wavetable declarations and wavetable placeholders in the instrument. All wavetables in the scope of the instrument may be referenced in a **tablemap**, regardless of the syntactic placement of the **tablemap**.

When the tablemap name is used in an array-reference expression (see subclause 5.8.6.7.5), the index of the expression determines to which of the wavetables in the list the expression refers. The first wavetable in the list is number 0, the second number 1, and so on.

EXAMPLE

For the following declarations

```
table t1(harm,2048,1);
imports table t2;
table t3(random,32,1);

tablemap tmap(t1,t2,t3,t2);
ivar i,x,y,z;
```

the following two code blocks are identical in semantics:

BLOCK 1

```
i = 3;
x = tableread(tmap[0],4);
y = tableread(tmap[i],3);
z = tableread(tmap[i > 4 ? 1 : 2],5);
```

BLOCK 2

```
x = tableread(t1,4);
y = tableread(t2,3);
z = tableread(t3,5);
```

Note that, like table references, array expressions using tablemaps may only occur in the context of an opcode or oparray call to an opcode accepting a wavetable reference.

5.8.6.6 Block of code statements

5.8.6.6.1 Syntactic form

<block> -> <statement> [<block>]
<block> -> **<NULL>**

<statement> -> <lvalue> **=** <expr> **;**
<statement> -> <expr> **;**
<statement> -> **if (** <expr> **) {** <block> **}**
<statement> -> **if (** <expr> **) {** <block> **} else {** <block> **}**
<statement> -> **while (** <expr> **) {** <block> **}**
<statement> -> **instr <ident> (** <expr list> **) ;**
<statement> -> **output (** <expr list> **) ;**
<statement> -> **spatialize (** <expr list > **) ;**
<statement> -> **outbus (<ident> ,** <expr list> **) ;**
<statement> -> **extend (** <expr> **) ;**
<statement> -> **turnoff ;**

<expr list> -> <expr> [**,** <expr list>]
<expr list> -> **<NULL>**

<lvalue> as given in subclause 5.8.6.6.2.
<expr> as given in subclause 5.8.6.7.

A block is a sequence of zero or more statements. A statement shall take one of several forms, which are enumerated and described in the subsequent subclauses. Each statement has rate-semantics rules governing the rate of the statement, the rate contexts in which it is allowable, and the times at which various subcomponents shall be executed.

To execute a block of statements at a particular rate, the statements within the block shall be executed, each at that rate, in such order as to produce equivalent results to executing the statements sequentially in linear order, according to the semantics below governing each type of statement.

5.8.6.6.2 Assignment

<statement> -> <lvalue> **=** <expr> **;**

<lvalue> -> **<ident>**
<lvalue> -> **<ident> [** <expr> **]**

<expr> as given in subclause 5.8.6.7.

An assignment statement calculates the value of an expression and changes the value of a signal variable or variables to match that value.

The lvalue, or left-hand-side value, denotes the signal variable or variables whose values are to be changed. An lvalue may be a local variable name, in which case the denotation is to the storage space associated with that name. An lvalue may also be a local array name, in which case the denotation is to the entire array storage space. An lvalue may also be a single element of a local array denoted by indexing a local array name with an expression. An lvalue shall not be a table reference or tablemap expression. An lvalue shall not be a standard name other than **MIDIctrl** or **params** (see subclause 5.8.6.8.9 and 5.8.6.8.26).

If the lvalue denotes an entire array, the right-hand-side expression of the assignment shall denote an array-valued expression with the same array length, or a single value, otherwise the construction is syntactically illegal. In the case that lvalue depends on the ordering of computation of the right-hand-side expression (e.g. if an entire array is multiplied with one of its elements), lvalue is undefined.

If the lvalue denotes a single value, the right-hand-side expression of the assignment shall denote a single value, otherwise the construction is syntactically illegal.

The rate of the lvalue is the rate of the signal variable, if there is no indexing expression, or the faster of the rate of the signal array denoted by the signal variable and the rate of the indexing expression, if there is an indexing expression.

The rate of the right-hand side is the rate of the right-hand-side expression.

The rate of the statement is the rate of the lvalue, however, the statement is illegal if the rate of the right-hand side is faster than the rate of the lvalue.

The assignment shall be performed as follows:

At every pass through the statement occurring at equal rate to the rate of the assignment, the right-hand side expression shall be evaluated. Then, the storage space denoted by the lvalue shall be updated to be equal to the value of the right-hand expression. If the lvalue denotes an entire array, and the right-hand-side expression a single value, then each of the values of each of the elements of the array shall be changed to the single right-hand-side value.

5.8.6.6.3 Null assignment

<statement> -> <expr> **;**

A null assignment contains only an expression; it is provided so that opcodes that do not have useful return values need not be used in the context of an assignment to a dummy variable.

The rate of the statement is the rate of the expression. The expression may be single-valued or array-valued; it shall not be a table reference.

The null assignment shall be performed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, the expression shall be evaluated.

5.8.6.6.4 If

<statement> -> **if (** <expr> **) {** <block> **}**

An **if** statement allows conditional evaluation of a block of code. The expression that is tested in the **if** statement is termed the guard expression.

The rate of the statement is the rate of the guard expression, or the rate of the fastest statement in the guarded code block, whichever is faster.

It is not permissible for the block of code governed by the **if** statement to contain statements slower than the guard expression. It is further not permissible for any of the statements in the governed block of code to contain calls to opcodes that would be executed slower than the guard expression. The guard expression shall be a single-valued expression.

EXAMPLE

The following code fragment contains a rate-mismatch error:

```
asig a; ksig k;

a = 0; if (a < 20) {
 k = kline(...);
}
```

The example is illegal because the **kline** assignment statement is slower than the guard **a < 20**. Even if the assignment were to an a-rate variable ("a2 = kline(...)"), thus making the assignment statement an a-rate statement, the example would be illegal, because the **kline** opcode itself is slower than the guard expression.

The **if** statement shall be executed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, the guard expression shall be evaluated. If the guard statement evaluates to any non-zero value in a particular pass, then the block of code shall be evaluated at the rate corresponding to that pass.

If a block of code executing at the **a-rate** or **k-rate** has **i-rate** statements, these statements should only be executed the first time the block executes, with regards to a particular state.

If a block of code executing at the **a-rate** has **k-rate** statements, these statements should only be executed the first time the block executes in a kcycle.

5.8.6.6.5 Else

<statement> -> **if (** <expr> **) {** <block> **} else {** <block> **}**

An **else** statement allows disjunctive evaluation of two blocks of code. The expression that is tested in the **else** statement is termed the guard expression.

The rate of the statement is the rate of the guard expression, or the rate of the fastest statement in the first guarded block of code, or the rate of the fastest statement in the second guarded block of code, whichever is fastest.

It is not permissible for the blocks of code governed by the **else** statement to contain statements slower than the guard expression. It is further not permissible for any of the statements in the governed blocks of code to contain calls to opcodes that would be executed slower than the guard expression. The guard expression shall be a single-valued expression.

The **else** statement shall be executed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, the guard expression shall be evaluated. If the guard expression evaluates to any non-zero value in a particular pass, then the first guarded block of code shall be at the rate corresponding to that pass. If the guard statement evaluates to zero in a particular pass, then each statement in the second guarded block of code shall be so evaluated.

If a block of code executing at the **a-rate** or **k-rate** has **i-rate** statements, these statements should only be executed the first time the block executes, with regards to a particular state.

If a block of code executing at the **a-rate** has **k-rate** statements, these statements should only be executed the first time the block executes in a kcycle.

5.8.6.6.6 While

<statement> -> **while (** <expr> **) {** <block> **}**

The **while** statement allows a block of code to be conditionally evaluated several times in a single rate pass. The expression that is tested in the **while** statement is termed the guard expression.

The rate of the **while** statement is the rate of the guard expression.

It is not permissible for the block of code governed by the **while** statement to contain statements that run at a rate other than the rate of the guard expression. It is further not permissible for any of the statements in the governed block of code to contain calls to opcodes that would be executed at a rate other than the rate of the guard

expression. The guard expression shall be a single-valued expression. It is not permissible for the guard expression to contain calls to core opcodes with type **specialop**, see subclause 5.9.2.

The **while** statement shall be executed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, the guard expression shall be evaluated. If the guard expression evaluates to any non-zero value in a particular pass, then each statement in the guarded block of code shall be evaluated according to the particular rules for that statement, and then the guard expression re-evaluated, iterating until the guard expression evaluates to zero.

5.8.6.6.7 Instr

<statement> -> **instr <ident> (** <expr list> **) ;**

The **instr** statement allows an instrument instantiation to dynamically create other instrument instantiations, for layering or synthetic-performance techniques. It shall consist of an identifier referring to an instrument defined in the current orchestra, a time delay, a duration, and a list of expressions defining parameters to pass to the referenced instrument.

It is a syntax error if the number of expressions in the expression list is not two greater than the number of pfields accepted by the referenced instrument (the first expression is the time delay and the second is the duration). Each expression in the expression list shall be a single-valued expression.

The rate of the **instr** statement is the rate of the fastest expression in the expression list, or the rate of the guarding expression containing the statement, or the rate of the opcode containing the statement, whichever is fastest.

It is not permissible for the rate of the **instr** statement to be a-rate.

The **instr** statement shall be executed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, each of the expressions in the expression list is evaluated. Then, a new instrument event is registered with the scheduler as described in subclause 5.11.3. The dispatch time of the new instrument event is the sum of the current orchestra time and the value of the first expression in the expression list, the latter scaled by the current global tempo; the duration of the new instrument event is the value of the second expression in the expression list; and the values of the p-fields for the new instrument event are the values of the remaining expressions in the expression list. To clarify, note that the values of the first and second expressions are considered to have units of score beats, not absolute time, and they are consequently scaled according to the actual **tempo** of the orchestra.

An exception to the above occurs when the time-delay (the first expression in the expression list) is less than the length of the orchestra control period. In this case, an instrument event is not created, but a new instrument instantiation is immediately created, where the duration of the new instantiation is the value of the second expression in the expression list, and the values of the instrument p-fields in the new instantiation are set to the values of the remaining expressions.

In this case, the i-rate pass through the new instrument instantiation shall be executed immediately upon its creation, before any more statements from the block of code containing the **instr** statement are executed. However, any changes to global i-rate variables made in the new instance during its i-rate pass are not respected in this instrument (the “caller”) (i-rate variables imported from the global context are set only during the initialisation pass of each instance, and never change afterward). The first k-rate and a-rate passes through the new instrument instantiation shall be executed as appropriate to the sequencing relation between the instantiating and instantiated instruments; that is, if the new instrument is sequenced later than the instantiating instrument, the new instantiation shall be executed at some later time in the same orchestra pass, but if the new instrument is sequenced earlier than the instantiating instrument, then the new instantiation shall not be executed in k-time or a-time until the subsequent orchestra pass.

A dynamically created instrument has access to the same **MIDIctrl** (subclause 5.8.6.8.9), **MIDItouch** (subclause 5.8.6.8.10), **MIDIbend** (subclause 5.8.6.8.11), **channel** (subclause 5.8.6.8.12) and **preset** (subclause 5.8.6.8.13)

standard name state as its parent. However, the dynamically created instrument is not scheduled for termination when the parent is terminated under MIDI control.

5.8.6.6.8 Output

<statement> -> **output (** <expr list> **) ;**

The **output** statement creates audio output from the instrument. This output does not get turned directly into sound, but rather gets buffered either on one or more busses based on instructions given in **route** statements (subclause 5.8.5.4) or on the special bus **output_bus** by default. However, if the current instrument instantiation is the one created with a **send** statement referencing the special bus **output_bus**, then the output of the current instantiation, created by summing its calls to **output**, may be turned directly into sound.

The expression list shall contain at least one expression.

The rate of the **output** statement is a-rate.

All statements within a orchestra that reference the same bus, whether through explicit sends, calls to **outbus**, or by default routing to the special bus **output_bus**, shall have compatible numbers of expression parameters representing output channels. “Compatible” means that if any calls to **output** for a particular bus reference more than one expression parameter, then all other calls to **output** referencing this bus shall have either the same number of expression parameters, or else only a single expression parameter. In addition, the number of channels of the special bus **output_bus** shall be the same as the global **outchannels** parameter and uses of **output** by instrument instances that are implicitly or explicitly routed to **output_bus** shall be compatible with this number of channels.

The **output** statement is executed as follows:

At each k-rate pass through the instrument, an output buffer, with number of channels determined by the rules in subclause 5.7.3.3.5.2, shall be cleared to zero values. At every a-rate pass through the statement, the expression parameters shall each be evaluated. Then, the expression parameter values shall be placed in the output buffer: if the **output** statement has more than one parameter expression, then the value of each parameter shall be added to the current value of the output buffer in the corresponding channel. If the **output** statement has only one parameter expression, then the value of that expression shall be added to the current value of the output buffer in each channel.

The expression parameters to the **output** statement may be array-valued, in which the mapping described in the preceding paragraph is not from expressions to buffer channels, but from array value channels to buffer channels.

EXAMPLE

The following code fragment

```
asig a[2], b;

. . .

output(a,b);
output(a[1],b,b);
output(b);
```

is legal and describes an instrument that outputs three channels of sound. The first channel of output contains the value `a[0] + a[1] + b`, the second `a[1] + b + b`, and the third `b + b + b`.

After each a-rate pass through the instrument instantiation during a particular orchestra pass, the values in the output buffer shall be added channel-by-channel to the current values of the bus or busses referenced by the **route** expression or expressions that also reference this instrument. If there are no such **route** statements, the values in the output buffer shall be added channel-by-channel to the current values of the special bus **output_bus**. If this is

the instrument instantiation created by referencing the special bus **output_bus** in a **send** statement, then the preceding two sentences do not hold, and instead the values in the output buffer are the output of the orchestra.

5.8.6.6.9 Spatialize

<statement> -> **spatialize (** <expr list > **) ;**

The **spatialize** statement allows instruments to produce spatialised sound, using non-normative methods that are implementation-dependent.

The expression list shall contain four expressions. The second, third, and fourth shall not be a-rate expressions. The first expression represents the audio signal to be spatialised; the second, the azimuth (angle) from which the source sound shall apparently come, measuring in radians clockwise from 0 azimuth directly in front of the listener; the third, the elevation angle from which the sound source shall apparently come, measuring in radians upward from 0 elevation on the listener's horizontal place; and the fourth, the distance from which the sound source shall apparently come, measuring in metres from the listener's position. Each of the four expressions shall be single-valued.

The rate of the **spatialize** statement is a-rate.

The **spatialize** statement shall be executed as follows:

At each a-rate pass through the instrument, the expressions in the expression list shall be evaluated. Then, the sound signal in the first expression shall be presented to the listener as though it has arrived from the azimuth, elevation, and distance given in the second, third, and fourth expressions. No normative requirements are placed on this spatialisation capability, although terminal implementers are encouraged to provide the maximum sophistication possible.

The sound produced via the **spatialize** statement is turned directly into orchestra output; it shall not be affected by bus routings or further manipulation within the orchestra. If multiple calls to **spatialize** occur within an orchestra, the various sounds so produced shall be mixed via simple summation after spatialisation. Similarly, if both spatialised and non-spatialised sound is produced within an orchestra, the final orchestra output of all non-spatialised sound shall be mixed via simple summation with the various spatialised sounds for presentation. The sound produced via each **spatialize** statement shall have as many channels as the global orchestra number of output channels (see subclause 5.8.5.2.4) in order to enable this mixing.

5.8.6.6.10 Outbus

<statement> -> **outbus (<ident> ,** <expr list> **) ;**

The **outbus** statement allows instruments to place dynamically-calculated signals on busses. The identifier parameter shall refer to the name of a bus defined with a **send** statement in the global block. The remaining expressions represent signals to place on the bus.

It is a syntax error if there are no expressions in the expression list, or if the identifier does not refer to a bus defined in the global block with a **send** statement. The number of expressions in the expression list shall be compatible with other statements making reference to the same bus, as defined in subclause 5.8.6.6.8.

The rate of the **outbus** statement is a-rate.

The **outbus** statement shall be executed as follows:

At each a-rate pass through the statement, the expression list shall be evaluated. Then, the expression values shall be added to the current values of the referenced bus. If there is more than one expression in the expression list, then each expression value shall be added to the corresponding channel of the referenced bus. If there is only one expression in the expression list, then the value of that expression shall be added to each channel of the referenced bus.

The expressions in the expression list may be array-valued, in which case the semantics are analogous to those in subclause 5.8.6.6.8.

The **outbus** statement shall not be used in an instrument that is the target of a **send** statement referencing the special bus **output_bus**. In addition, an **outbus** statement may not write to the special bus **input_bus**.

5.8.6.6.11 Extend

<statement> -> **extend (** <expr> **) ;**

The **extend** statement allows an instrument instantiation to dynamically lengthen its duration.

The expression parameter shall not be a-rate. The expression shall be single-valued.

The rate of the **extend** statement is the rate of the expression parameter.

The **extend** statement shall be executed as follows:

At each pass through the statement at equal rate to the rate of the statement, the expression shall be evaluated. Then, the duration of the instrument instantiation shall be extended by the amount of time, in seconds, given by the value of the expression. That is, if the instrument instance had been previously scheduled to be terminated at time *t*, then after a call to **extend** with an expression evaluating to *s*, the instrument instance shall be scheduled to terminate at time *t*+*s*. If the instrument instance had no scheduled termination time (its duration was –1 on instantiation), **extend** with an expression evaluating to *s* shall schedule termination of the instrument at time $T + s$, where *T* is the current orchestra time.

The **extend** statement shall not be executed in an instrument instance that is created as the result of a send statement referencing the special bus **output_bus**.

NOTE - The parameter of **extend** is specified in seconds, not in beats. If it is desirable to have time-extension dependant on tempo in a particular composition, the content author may enable this by rescaling the parameter by the current value of **gettempo()** (subclause 5.9.15.1).

extend may be called with a negative argument to shorten the duration of a note; if $t+s < T$ (that is, if the negatively extended duration has already been exceeded in the instantiation), then the statement acts as the **turnoff** statement, see subclause 5.8.6.6.12.

When the **extend** statement is called, the standard name **dur** shall be updated to reflect the new duration; that is, **dur := dur + x** where **x** is the expression value of the parameter.

5.8.6.6.12 Turnoff

<statement> -> **turnoff ;**

The **turnoff** statement allows an instrument instantiation to dynamically decide to terminate itself.

The rate of the **turnoff** statement is k-rate.

The **turnoff** statement shall be executed as follows:

When the **turnoff** statement is reached at k-rate, the instrument instance shall be scheduled to terminate after the following k-cycle; that is, if the current orchestra time is *T* and the k-pass duration *k*, the instrument instantiation shall be scheduled to terminate at time *T*+*k*.

The **turnoff** statement shall not update the **dur** standard name.

The **turnoff** statement shall not be executed in an instrument instance that is created as the result of a **send** statement referencing the special bus **output_bus**.

NOTE - **turnoff** does not destroy the instantiation immediately; the instantiation is executed for one more orchestra pass, to allow the instrument time to examine the **released** variable. Instruments may call **turnoff** and then "save" themselves on the subsequent k-cycle by calling **extend**.

5.8.6.7 Expressions

5.8.6.7.1 Syntactic form

<expr> -> **<ident>**
<expr> -> **<number>**
<expr> -> **<int>**
<expr> -> **<ident> [** <expr> **]**
<expr> -> **<ident> (** <expr list> **)**
<expr> -> **<ident> [** <expr> **] (** <expr list> **)**
<expr> -> <expr> **?** <expr> **:** <expr>
<expr> -> <expr> <binop> <expr>
<expr> -> **!** <expr>
<expr> -> **-** <expr>
<expr> -> **(** <expr> **)**
<expr> -> **sasbf (** <expr list> **) ;**

<binop> -> **+**
<binop> -> **-**
<binop> -> *
<binop> -> **/**
<binop> -> **==**
<binop> -> **>=**
<binop> -> **<=**
<binop> -> **!=**
<binop> -> **>**
<binop> -> **<**
<binop> -> **&&**
<binop> -> **||**

An expression can take one of several forms, the semantics of which are enumerated in the subclauses below. Each form has both rate semantics, which describe the rate of the expression in terms of the rates of the subexpressions, and value semantics, which describe the value of the expression in terms of the values of the subexpressions. The syntax above is ambiguous for many expressions; disambiguating precedence rules are given in subclause 5.8.6.7.14.

5.8.6.7.2 Properties of expressions

Each expression is conceptually labelled with two properties: its rate and its width. The rate of an expression determines how fast the value of that expression might change; the width of an expression determines how many channels of sound or other data are represented by the expression. In each expression type, the rate and width of the expression are determined from the type of the expression, and perhaps from the rate and width of the component subexpressions.

NOTE - Any name declared as an array is an array-valued variable regardless of its length. That is, a variable declared as **asig name[1]** is not a single-valued variable.

5.8.6.7.3 Identifier

<expr> -> **<ident>**

An identifier expression denotes a storage location or locations that contain values stored in memory. It is illegal to reference an identifier that is not declared in the local instrument or opcode scope, and that is not a standard name (see subclause 5.8.6.7.14).

The rate of an identifier expression is the rate type at which the identifier was declared, or is implicitly declared in the case of standard names. The rate of a **table** identifier is i-rate.

If the identifier denotes a single-valued name (i.e., one that is not an array type), then the value of the identifier expression is the value stored in memory associated with that identifier in the current scope, and the width of the expression is 1.

If the identifier denotes an array-valued name, then the value of the identifier expression is the ordered sequence of values stored in memory and associated with that identifier in the current scope, and the width of the expression is the width of the array so denoted.

If the identifier denotes a table, then the value of the identifier expression is a reference to the table with the given name. Table references may only appear in calls to opcodes. A table reference has width 1.

5.8.6.7.4 Constant value

<expr> -> **<number>**
<expr> -> **<int>**

A constant value expression denotes a single number.

The rate of a constant value expression is i-rate.

The width of a constant value expression is 1.

The value of a constant expression is the value of the number denoted by the constant. The value of a constant expression is always a floating-point value, whether the token or lexical expression denoting the value was an integer or floating-point token or expression.

5.8.6.7.5 Array reference

<expr> -> **<ident> [** <expr> **]**

An array reference expression allows the selection of one value from an array of several. The identifier in the array-reference syntax is termed the array name, and the expression the index expression. It is illegal to use an identifier in an array reference that is neither declared in the local instrument or opcode scope as an array, nor implicitly defined as an array-valued standard name or table map.

The index expression shall have width 1.

The rate of an array reference expression is the rate of the array name (which is the rate at which the array name was declared explicitly or implicitly), or the rate of the index expression, whichever is faster.

The width of an array reference expression is 1.

If the referenced array is an array-valued signal variable, then the value of the array reference expression is the value of that element of the sequence of values in the array storage corresponding to the value of the indexing expression, where element 0 corresponds to the first value in the sequence. It is a run-time error if the value of the indexing expression is less than 0, or equal to or greater than the declared size of the array. If the indexing expression is not an integer, it is rounded to the nearest integer.

If the referenced array is a table map, then the value of the array reference expression is a reference to that element of the sequence of tables corresponding to the value of the index expression, where element 0 corresponds to the first table in the sequence. It is a run-time error if the value of the indexing expression is less

than 0, or equal to or greater than the declared size of the table map. If the indexing expression is not an integer, it is rounded to the nearest integer. Table references may only appear in calls to opcodes. See also the example in subclause 5.8.6.5.6.

NOTE - The syntax `t[i]`, where **t** is a table rather than a table map, is illegal. The **tableread** core opcode is used to directly access elements of a wavetable. See subclause 5.9.6.

5.8.6.7.6 Opcode call

<expr> -> **<ident> (** <expr list> **)**

An opcode call expression allows the use of processing functionality encapsulated within an opcode.

The identifier is termed the opcode name, and the expression list the actual parameters of the opcode call expression. It is illegal to use an identifier that is not the name of a core opcode and is also not the name of a user-defined opcode declared elsewhere in the orchestra. For user-defined opcodes, the number of actual parameters shall be the same as the number of formal parameters in the opcode definition. For core opcodes without variable argument lists, the number of actual parameters required varies from opcode to opcode; see subclause 5.8.9. If a particular formal parameter in an opcode definition is an array, then the corresponding actual parameter shall be an array-typed expression of equal width. If a particular formal parameter in an opcode definition is a table, then the corresponding actual parameter shall be a table reference.

If a particular formal parameter in an opcode definition is at a particular rate, then the corresponding actual parameter expression shall not be at a faster rate.

The rate of the opcode call expression is determined according to the rules in subclause 5.8.7.7.

The width of the opcode call expression is the number of channels provided in the **return** statements in the opcode's code block.

For calls to core opcodes (see subclause 5.9), in the absence of normative language specifying otherwise for a particular opcode, it is a syntax error if any of the following statements apply:

- there are fewer actual parameters in the opcode call than required formal parameters
- there are more actual parameters in the opcode call than required and optional formal parameters, and the opcode definition does not include a varargs "..." clause.
- a particular actual parameter expression is of faster rate than the corresponding formal parameter, or than the varargs formal parameter if that is the correspondence
- a particular actual parameter expression is not single-valued, or is not table-valued when the corresponding formal parameter specifies a table.

The context of the opcode call is restricted more than other expressions. When occurring within a block subsidiary to a guarding statement (**if**, **else**, or **while**), opcode calls shall not have a rate slower than the rate of the guarding expression (see subclauses 5.8.6.6.4 and 5.8.6.6.5 and 5.8.6.6.6). A call to an opcode with a particular name shall not occur within the code block of definition of that opcode, nor within the code blocks of any of the opcodes called by that opcode, or any of the opcodes called by them, etc. That is, recursive and mutually-recursive opcodes are prohibited.

If a **kopcode** opcode call occurs in a expression that runs at the **a-rate**, the first time this expression is executed in a k-cycle with regards to a particular opcode state, the **kopcode** is called, following the semantics described in this subclause. For all subsequent evaluations of the expression in the same k-cycle, the **kopcode** is not executed; instead, the return value from the first execution is used in the expression evaluation.

If an **iopcode** opcode call occurs in a expression that runs at the **a-rate** or the **k-rate**, the first time this expression is executed with regards to a particular opcode state, the **iopcode** is called, following the semantics described in

this subclause. For all subsequent evaluations of the expression, the **iopcode** is not executed; instead, the return value from the first execution is used in the expression evaluation.

If a **specialop** core opcode call occurs in a expression that runs at the **a-rate**, the **k-rate** semantics of the **specialop** opcode call follows the rules for **kopcode** calls described above, while the **a-rate** semantics of the **specialop** opcode call happen at every a-cycle as described in subclause 5.9.2.

To calculate the value of an opcode call expression referencing a user-defined opcode at a particular rate, the values of the actual parameter expression shall be calculated in the order they appear in the expression list. The values of the formal parameters within the opcode scope shall be set to the values of the corresponding actual parameter expressions. If this is the first opcode call expression referencing this opcode scope, opcode storage space shall be created to store local signal variables and wavetables, the local signal variables set to 0, and the local wavetables created as discussed in subclause 5.8.6.5.2. Any global variables imported by the opcode at that rate shall be copied into the opcode storage space. The statement block of the opcode shall be executed at the current rate. The value of the opcode call expression is the value of the first **return** statement encountered when executing the opcode. The value of the opcode call expression may be array-valued (if the expression in the **return** statement is). After the end of opcode execution, any global variables exported by the opcode shall be copied into the global storage space.

NOTE - If an opcode changes and exports the value of a global variable that is imported by the calling instrument or opcode, the change in the global variable is not reflected in the caller until the next orchestra pass.

If a particular actual parameter expression in an opcode call expression is an identifier or an array-reference expression, then that parameter is a reference parameter in that call to that opcode. When the opcode statement block is executed, the final value of the formal parameter associated with that actual parameter shall be copied into the variable value denoted by the identifier or array-reference, unless the parameter is a standard name which may not be used as an lvalue (see subclause 5.8.6.6.2), in which case no copy is performed. This modification shall happen immediately after (but not until) the termination of the statement block, before any other calculation is done. Both single-value and array-value expressions may be reference parameters, but if an array-valued expression is used, the associated formal parameter shall be an array of the same length.

To calculate the value of an opcode call expression referencing a core opcode at a particular rate, the values of the actual parameter expressions shall be calculated in the order they appear in the expression list. Then, the return value of the core opcode shall be calculated according to the rules for the particular opcode given in subclause 5.8.9. If the rate of a formal parameter is slower than the rate of the opcode, the following rules apply:

- in an opcode call at the **k-rate**, actual variables associated to an **ivar** formal parameters are updated only the first time this opcode is executed with regards to a particular opcode state;

- in an opcode call at the **a-rate**, actual variables associated to an **ivar** formal parameters are updated only the first time this opcode is executed with regards to a particular opcode state; actual variables associated to a **ksig** formal parameters are updated only the first time this opcode is executed in a k-cycle with regards to a particular opcode state.

NOTE - The variables declared within the scope of a user-defined opcode are static-valued; that is, they preserve their values from call to call. The values of variables within the scope of a user-defined opcode are set to 0 before the opcode is called the first time. Each syntactically distinct call to an opcode creates one and only one opcode scope (see example in next subclause).

5.8.6.7.7 Oparray call

<expr> -> **<ident> [** <expr> **] (** <expr list> **)**

An oparray call expression allows the dynamic selection of an opcode state from a set of several, and the calculation of encapsulated functionality with respect to that opcode state.

The identifier is termed the opcode name, the expression in brackets is termed the index expression, and the expressions in the parameter list are termed the actual parameters. It is illegal to use an identifier that is not the name of a core opcode and is also not the name of a user-defined opcode declared elsewhere in the orchestra. It

is also illegal to use an identifier for which oparray storage is not allocated in the local scope as described in subclause 5.8.6.5.5. For user-defined opcodes, the number of actual parameters shall be the same as the number of formal parameters in the opcode definition. For core, the number of actual parameters required varies from opcode to opcode; see subclause 5.8.9.

The index expression shall be a single-valued expression.

The rate of the oparray call expression is the rate of the opcode referenced, as determined by the rules in subclause 5.8.7. The rate of the index expression shall not be faster than the rate of the opcode referenced.

The width of the oparray call expression is the number of channels returned by **return** statements within the opcode code block.

The context of the oparray call expression is restricted in the same way as described for the opcode call expression in subclause 5.8.6.7.6. The rate semantics for oparray call execution follows the same rules described for the opcode call expression in subclause 5.8.6.7.6.

The value of the oparray call expression is determined in the same way as described for the opcode call in subclause 5.8.6.7.6, with the following exceptions and additions:

Before the values of the actual parameter expressions are calculated, the value of the index expression is calculated. It is a run-time error if the value of the index expression is not in the range [0..n-1], where n is the allocation size in the oparray definition for this oparray. If the index expression is not an integer, it is rounded to the nearest integer. The scope storage associated with the opcode name and the value of the index expression is selected from the set of oparray scopes in the local scope. The evaluation of the statement block in the referenced opcode is with regard to the selected scope. Within each oparray scope, local variables retain their values from call to call.

EXAMPLES

Some examples are provided to clarify the distinction between opcode calls and oparray calls.

The following user defined opcode

```
kopcode inc() {
  ksig ct;

  ct = ct + 1;
  return(ct);
  }
```

counts the number of times it is called.

1. After the first execution of the following code fragment

```
a = inc();
b = inc();
```

the value of **a** is 1, and the value of **b** is 1, since each call to **inc()** refers to a different scope.

2. After the first execution of the following code fragment

```
i = 0; while (i < 2) { a = inc(); i = i + 1; }
```

the value of **a** is 2, since there is only one scope for **inc().**

3. After the first execution of the following code fragment

```
oparray inc[2];
```

```
a = inc[0]();
b = inc[0]();
```

the value of **a** is 1, and the value of **b** is 2, since each call to **inc()** refers to the same scope (since the value of the indexing expression is the same in both calls).

4. After the first execution of the following code fragment

```
oparray inc[2];

i = 0; while (i < 2) { a = inc[i](); i = i + 1; }
```

the value of **a** is 1, since each iteration refers to a different scope in the call to **inc()** (since the value of the indexing expression is 0 on the first iteration, and 1 on the second).

NOTE - Opcode calls and oparray calls referencing the same opcode may be used in the same scope. In this case, the scopes referenced by each of the opcode calls are different from any of the scopes defined in the oparray definition.

5.8.6.7.8 Combination of vector and scalar elements in mathematical expressions

The subsequent subclauses (subclauses 5.8.6.7.9 through 5.8.6.7.13) describe mathematical expressions in SAOL. For each, the width of the expression is the maximum width of any of its subexpressions. For each expression type, each subexpression within an expression shall have the same width, or else width of 1. If subexpressions with width 1 and width different than 1 are combined in an expression, before the expression is computed, the subexpression(s) with width 1 shall be promoted to have the same width as the expression. That is, a width 1 expression with value **x** that is a subexpression of a width *n* expression shall be promoted to a width *n* expression where the value of each element is **x**.

For each expression type below, the semantics will be given for array-valued expressions. In each case, the semantics for the single-valued expression are the same as for an array-valued expression with width 1, except for the special cases of switch, logical AND, and logical OR, which will be described separately in those subclauses.

5.8.6.7.9 Switch

<expr> -> <expr> **?** <expr> **:** <expr>

The switch expression combines values from two subexpressions based on the value of a third.

The rate of the switch expression is the rate of the fastest of the three subexpressions.

The value of the switch expression is calculated as follows: the three subexpressions are evaluated. Then, for each value of the first subexpression, if this value is non-zero, the corresponding value of the switch expression is the corresponding value of the second subexpression. If this value is zero, the corresponding value of the switch expression is the corresponding value of the third subexpression.

In the special case where all subexpressions have width 1, then the switch expression “short-circuits”: the first subexpression is evaluated, and if its value is non-zero, then the second subexpression is evaluated, and its value is the value of the switch expression. If the value of the first subexpression is zero, then the third subexpression is evaluated, and its value is the value of the switch expression. If the width of the switch expression is 1, then in no case are both the second and third subexpressions evaluated.

5.8.6.7.10 Not

<expr> -> **!** <expr>

The not expression performs logical negation on a subexpression.

The rate of the not expression is the rate of the subexpression.

The value of the not expression is calculated as follows: the subexpression is evaluated. For each nonzero value in the subexpression, the corresponding value of the not expression is zero; for each zero value in the subexpression, the corresponding value of the not expression is 1.

5.8.6.7.11 Negation

<expr> -> **-** <expr>

The negation expression performs arithmetic negation on a subexpression.

The rate of the negation expression is the rate of the subexpression.

The value of the negation expression shall be calculated as follows: the subexpression is evaluated. For each value in the subexpression, the corresponding value of the negation expression is the arithmetic negative of the value.

5.8.6.7.12 Binary operators

<expr> -> <expr> <binop> <expr>

There are 12 binary operators. Each of them calculates a different function on binary subexpressions.

The value of the expression shall be calculated as follows. The two subexpressions shall be evaluated, and for each pair of values of the subexpressions, and the corresponding value of the binary expression shall be calculated according to the following table, where x_1 and x_2 are the values of the first and second subexpressions:

Table 5.2 — Binary operators

Operator	Value of expression
+	$x_1 + x_2$
-	$x_1 - x_2$
*	$x_1 x_2$
/	x_1 / x_2
==	if $x_1 = x_2$, then 1, otherwise 0
>	if $x_1 > x_2$, then 1, otherwise 0
<	if $x_1 < x_2$, then 1, otherwise 0
<=	if $x_1 \leq x_2$, then 1, otherwise 0
>=	if $x_1 \geq x_2$, then 1, otherwise 0
!=	if $x_1 \neq x_2$, then 1, otherwise 0

In each of these cases, if the particular operation would result in a NaN or Inf result (for example, division by 0), a run-time error shall result.

For the “logical and” operator **&&** in the special case where both subexpressions have width 1, the expression is calculated in a “short-circuit” fashion. The first subexpression shall be evaluated. If its value is 0, then the value of the expression is 0; if its value is nonzero, then the second subexpression shall be evaluated, and if its value is 0, then the value of the expression is 0, otherwise the value of the expression is 1.

For the “logical or” operator || in the special case where both subexpressions have width 1, the expression is calculated in a “short-circuit” fashion. The first subexpression shall be evaluated. If its value is nonzero, then the value of the expression is 1; if its value is 0, then the second subexpression shall be evaluated, and if its value is nonzero, then the value of the expression is 1, otherwise the value of the expression is 0.

5.8.6.7.13 Parenthesis

<expr> -> **(** <expr> **)**

The parenthesis operator performs no new calculation, but allows the specification of arithmetic grouping.

The rate of the parenthesis expression is the rate of the subexpression.

The width of the parenthesis expression is the width of the subexpression.

The value of the parenthesis expression is the value of the subexpression.

5.8.6.7.14 Order of operations

Expressions bind in the order prescribed in the following table. That is, operations listed higher in the table are performed before operations lower in the table whenever the ordering is syntactically ambiguous. Operations listed on the first and last row associate right-to-left, that is, the rightmost expression is performed first. Operations listed on the remaining rows associate left-to-right, that is, the leftmost expression is performed first.

Table 5.3 — Order of operations

Operator	Function
!, -	not, unary negation
*, /	multiply, divide
+, -	add, subtract
<, >, <=, >=	relational
==, !=	equality
&&	logical and
||	logical or
?:	switch

5.8.6.7.15 SASBF synthesis

<expr> -> **sasbf (** <expr list> **) ;**

The **sasbf** expression allows the use of the DLS-compatible bank synthesis procedure (see subclause 5.13) within a SAOL instrument. It shall not be used in a Object type 3 bitstream and capability for executing it does not have to be provided by a Object type 3 decoder. The parameter list shall have two, three, or four expressions. All shall be single-valued i-rate expressions.

1. The first expression shall correspond to the MIDI pitch desired for synthesis. If this value is not an integer, it shall be rounded to the nearest integer. It is a run-time error if this value is less than 1 or greater than 128.

2. The second expression shall correspond to the MIDI velocity desired for synthesis. If this value is not an integer, it shall be rounded to the nearest integer. It is a run-time error if this value is less than 0 or greater than 128.

3. The third expression, if given, corresponds to the MIDI preset number. If there are less than three expressions, the MIDI preset number is the default value given by **p** mod 128, where **p** is the preset number of the instrument to which the MIDI noteon that created the note instance was directed, or the lowest-numbered instrument preset number for the instrument containing this statement (subclause 5.8.6.4), if the note was not triggered with a MIDI event.

4. The fourth expression, if given, corresponds to the MIDI bank number. If there are less than four expressions in the list, the MIDI bank number is the default value given by floor(**p/128)** + 1, where **p** is the preset number to which the MIDI noteon that created the note instance was directed, or the lowest-numbered instrument preset number for the instrument (subclause 5.8.6.4), if the note was not triggered by a MIDI event. It is a syntax error if there are less than four expressions and no instrument preset number is provided.

The **sasbf** expression is an a-rate expression. The **sasbf** expression has two channels defined by the bank-synthesis procedure as described in subclause 5.13.

The value of the **sasbf** expression is calculated as follows:

On the first execution of the expression, each expression in the expression list shall be evaluated. Using these values, one note of synthesis shall be dispatched to the wavetable bank synthesis procedure described in subclause 5.13.

On the first and each subsequent a-rate pass through the **sasbf** expression, the value of the expression shall be the next pair of audio samples from the stereo wavetable bank synthesis process.

During the wavetable bank synthesis process, the values of MIDI controllers and other continuous values on the current channel and note shall be respected. These values are not passed into the **sasbf** expression list, but are made available to the SASBF synthesiser in an implementation-dependent way. If the values of the global MIDIctrl[] standard name are changed by any instrument (see subclause 5.8.6.8.9), the new values shall be respected by all SASBF synthesis processes.

NOTE - Certain MIDI instructions (such as the Registered Parameter Number mechanism, see [MIDI]), cannot be properly understood on an event-by-event basis; rather, their semantics are understood to take effect when the entire RPN change is completed. To ensure this, all control events, whether or not they have SAOL semantics (see subclause 5.14.3.2), shall be passed through the SAOL scheduler to the **sasbf** synthesis processes in their original order.

If the instrument instance containing a particular **sasbf** expression was not instantiated in response to a MIDI event, then that instance is on no channel, and so the SASBF synthesis for that expression cannot be controlled by MIDI-based continuous controllers.

Each syntactically different instance of the **sasbf** expression results in one instance of a SASBF note synthesis procedure. There is no mechanism for interleaving samples from a single **sasbf** expression in multiple lines, or for instantiating multiple bank-synthesis procedures with one syntactic expression. **sasbf** is not an opcode and is not permitted to be used as an oparray construction (see subclause 5.8.6.7.7).

The value of the **released** standard name in the instrument instance (see subclause 5.8.6.8.16) containing the call to **sasbf** shall be made available to each SASBF process in an implementation-dependent way. The SASBF process shall use this flag to determine when to begin synthesis of the release portion of the given note. If a particular SASBF instance needs to extend the duration past the release time, it shall extend the note by one k-cycle in the manner of the **extend** statement (subclause 5.8.6.6.11). If, on the next k-cycle, the SASBF instance is still not finished, it may extend the note by a further k-cycle, and so on.

If multiple SASBF instances in an instrument each require extended duration, together they shall extend the duration by one k-cycle; the duration shall not be extended by one k-cycle per SASBF instance.

The SASBF synthesis process for each note terminates when the instrument instance containing this **sasbf** expression is destroyed. There is no mechanism for ending the SASBF synthesis earlier than this.

EXAMPLE

The following instrument layers two notes using the wavetable synthesiser and filters the result.

```
instr layer(mp, vel) preset 12 {
  asig a1[2], a2[2], i;
  oparray bandpass[2];

  a1 = sasbf(mp,vel);
  a2 = a1 + sasbf(mp,vel,47,2) / 2;   // second note quieter
  i = 0;
  while (i < 2) {
    a2[i] = bandpass[i](a2[i],400,100);
    i = i + 1;
    }
```

```
  output(a2);
}
```

The two instances of **sasbf** operate simultaneously and in parallel. The first synthesises sound from preset 13, bank 1 (since this is the preset number to which the instrument responds); the second, from preset 48, bank 2. They return audio signals that are summed together and manipulated with the **bandpass** core opcode.

5.8.6.8 Standard names

5.8.6.8.1 Definition

Not all identifiers to be referenced in an instrument or opcode are required to be declared as variables. Several identifiers, listed in this subclause, are termed standard names, shall not be used as variables, and have fixed semantics that shall be implemented in a compliant SAOL decoder. Standard names may otherwise be used as variables, embedded in expressions, etc. in any SAOL instrument or opcode. However, the semantics of using a standard name as an lvalue are undefined.

The implicit definition of each standard name, showing the rate semantics and width of that standard name, is listed, and the semantics of the value of the standard name specified in the subsequent subclauses.

5.8.6.8.2 k_rate

```
ivar k_rate
```

The standard name **k_rate** shall contain the control rate of the orchestra, in Hz.

5.8.6.8.3 s_rate

```
ivar s_rate
```

The standard name **s_rate** shall contain the sampling rate of the orchestra, in Hz.

5.8.6.8.4 inchan

```
ivar inchan
```

The standard name **inchan**, in each scope, shall contain the number of channels of input being provided to the instrument instantiation with which that scope is associated. “Associated” shall be taken to mean, for instrument code, the instrument instantiation for which the scope memory was created; for opcode code, the instrument instantiation that called the opcode, or called the opcode’s caller, etc.

Different instances of the same instrument may have different numbers of input channels if, for example, they are the targets of different send statements. Instructions for calculating the value of this standard name are provided in subclause 5.7.3.3.5.2

5.8.6.8.5 outchan

```
ivar outchan
```

The standard name **outchan**, in each scope, shall contain the number of channels of output provided by the instrument instantiation with which that scope is associated, in the sense described in subclause 5.8.6.8.4.

5.8.6.8.6 time

```
ivar time
```

The standard name **time**, in each scope, shall contain the time at which the instrument instantiation associated with that scope was created.

NOTE - If the “event time” of an instrument (for example, a score event more precisely timed than one control period) and the actual instantiation time differ, the name time shall contain the latter time, not the former.

5.8.6.8.7 dur

```
ivar dur
```

The standard name **dur**, in each scope, shall contain the duration of the instrument instantiation as originally created, and as potentially revised by use of the **extend** statement (subclause 5.8.6.6.11), or –1 if the duration was not known at instantiation.

Although **dur** is an i-rate variable, it may be changed during the duration of an instrument instance through the **extend** statement or through tempo changes. In this case, the value of expressions at the **ivar** do not change; expressions are only evaluated according to the rules in subclause 5.8.6.6.

5.8.6.8.8 itime

```
ksig itime
```

The standard name **itime**, in each scope, shall contain the elapsed time of the instrument instance. That is, on the first k-cycle through an instrument, **itime** shall be 0, and thereafter shall be incremented by 1/**KR**, where **KR** is the orchestra sampling rate, at the beginning of each k-rate pass.

5.8.6.8.9 MIDIctrl

```
ksig MIDIctrl[128]
```

The **MIDIctrl** standard variable shall contain, for each scope, the current values of the MIDI controllers on the channel corresponding to the channel to which the instrument instantiation associated with that scope is assigned. See subclause 5.13 for more details on MIDI control of orchestras.

Instruments may use **MIDIctrl** as an lvalue, that is, to assign new values to it using the **=** statement (subclause 5.8.6.6.2). In this case, when an instrument assigns to **MIDIctrl**, the value for the indicated controller shall be changed on the channel to which the instrument instance associated with that scope is assigned. The value of **MIDIctrl** is changed in all other instrument instances associated with that channel to the new value, and this change shall take effect the next time each of these instrument instances is executed at the k-rate (see subclause 5.7.3.3.6, list item 10).

MIDIctrl[64] is a special controller. It is normatively defined as the sustain controller, and has a special relationship with the MIDI **noteoff** instruction, see subclause 5.14.3.2.4.

5.8.6.8.10 MIDItouch

```
ksig MIDItouch
```

The **MIDItouch** standard variable shall contain, for each scope, the current value of the MIDI aftertouch on the note that caused the associated instrument instantiation to be created. See subclause 5.13 for more details on MIDI control of orchestras.

5.8.6.8.11 MIDIbend

```
ksig MIDIbend
```

The **MIDIbend** standard variable shall contain, for each scope, the current value of the MIDI pitchbend on the channel corresponding to the channel to which the instrument instantiation associated with that scope is assigned.

5.8.6.8.12 channel

```
ivar channel
```

The **channel** standard name contains the extended MIDI channel of the note responsible for creating the current instrument instance. See subclause 5.14.3.2.2.

5.8.6.8.13 preset

```
ivar preset
```

The **preset** standard name contains the SAOL preset number (which is one less than the MIDI preset number) of the note responsible for creating the current instrument instance. The **preset** standard name does not contain "all" of the preset numbers for the current instrument, only the one that led to the instantiation of the current instance.

5.8.6.8.14 input

```
asig input[inchannels]
```

The **input** standard variable shall contain, for each scope, the input signal or signals being provided to the instrument instantiation through the **send** instruction. See subclause 5.8.5.5.

5.8.6.8.15 inGroup

```
ivar inGroup[inchannels]
```

The **inGroup** standard variable shall contain, for each scope, the grouping of the input signals being provided to the instrument instantiation. See subclause 5.8.5.5.

5.8.6.8.16 released

```
ksig released
```

The **released** standard name shall contain, for each scope, 1 if and only if the instrument instantiation associated with the scope is scheduled to be destroyed at the end of the current orchestra pass. Otherwise, **released** shall contain 0. See subclause 5.7.3.3.6, list item 3.

5.8.6.8.17 cpuload

```
ksig cpuload
```

The **cpuload** standard name shall contain, for each scope, a measure of the recent CPU load on the CPU most strongly associated with the instrument instantiation associated with the scope. If the instrument instantiation is running entirely on one CPU, then that CPU shall be measured; if the instrument instantiation is running on multiple CPUs, then the exact measurement procedure is nonnormative.

The measure of CPU load shall be as a percentage of real-time capability: if the CPU is entirely loaded and cannot perform any more calculations without slipping out of real-time performance, the value of **cpuload** shall be 1 on that CPU at that k-cycle. If the CPU is entirely unloaded and is not performing any calculations, the value of **cpuload** shall be 0 on that CPU at that k-cycle. If the CPU is half-loaded, and could perform twice as many calculations in real-time as it is currently performing, the value of **cpuload** shall be 0.5 on that CPU at that k-cycle.

The exact calculation method, time window, recency, etc. of the CPU load is left to implementers.

5.8.6.8.18 position

```
imports ksig position[3]
```

The **position** name contains the absolute position of the node responsible for creating the current orchestra in the BIFS scene graph (see ISO/IEC 14496-1 subpart 9, particularly subclause 9.4.2.82). The position is given by the current value of the **position** field of the **Sound** node that is the ancestor of this node in the scene graph, as

transformed by its ancestors (that is, the final position in world co-ordinates of the **Sound** node). The value is global and shared by all instruments; it may not be changed by the orchestra.

5.8.6.8.19 direction

```
ksig direction[3]
```

The **direction** name contains the orientation of the node responsible for creating the current orchestra in the BIFS scene graph (see ISO/IEC 14496-1 clause 9, particularly subclause 9.4.2.82). The direction is given by the current value of the **direction** field of the **Sound** node that is the ancestor of this node in the scene graph, as transformed by its ancestors (that is, the final direction in world co-ordinates of the **Sound** node). The value is global and shared by all instruments; it may not be changed by the orchestra.

5.8.6.8.20 listenerPosition

```
ksig listenerPosition[3]
```

The **listenerPosition** name contains the absolute position of the listener in the BIFS scene graph (see ISO/IEC 14496-1 subpart 9, particularly subclause 9.4.2.82). The position is given by the current value of the **position** field of the active **ListeningPoint** node in the scene graph, as transformed by its ancestors (that is, the final position in world co-ordinates of the **ListeningPoint** node).

5.8.6.8.21 listenerDirection

```
ksig listenerDirection[3]
```

The **listenerDirection** name contains the orientation of the listener in the BIFS scene graph (see ISO/IEC 14496-1 subpart 9, particularly subclause 9.4.2.82). The direction is given by the current value of the **direction** field of the active **ListeningPoint** node in the scene graph, as transformed by its ancestors (that is, the final direction in world co-ordinates of the **ListeningPoint** node).

5.8.6.8.22 minFront

```
ksig minFront
```

The **minFront** standard name gives one parameter of the sound radiation pattern of the sound that the current node is a part of. This parameter, and its semantics, are defined by the **minFront** field of the **Sound** node of which this node is an ancestor (see ISO/IEC 14496-1 subpart 9, particularly subclause 9.4.2.82).

5.8.6.8.23 maxFront

```
ksig maxFront
```

The **maxFront** standard name gives one parameter of the sound radiation pattern of the sound that the current node is a part of. This parameter, and its semantics, are defined by the **maxFront** field of the **Sound** node of which this node is an ancestor (see ISO/IEC 14496-1 subpart 9, particularly subclause 9.4.2.82).

5.8.6.8.24 minBack

```
ksig minBack
```

The **minBack** standard name gives one parameter of the sound radiation pattern of the sound that the current node is a part of. This parameter, and its semantics, are defined by the **minBack** field of the **Sound** node of which this node is an ancestor (see ISO/IEC 14496-1 subpart 9, particularly subclause 9.4.2.82).

5.8.6.8.25 maxBack

```
ksig maxBack
```

The **maxBack** standard name gives one parameter of the sound radiation pattern of the sound that the current node is a part of. This parameter, and its semantics, are defined by the **maxBack** field of the **Sound** node of which this node is an ancestor (see ISO/IEC 14496-1 subpart 9, particularly subclause 9.4.2.82).

5.8.6.8.26 params

```
imports exports ksig params[128]
```

The **params** standard name is shared globally by all instruments. At each k-cycle of the orchestra, it shall contain the current values of the **params** field of the BIFS **AudioFX** node responsible for instantiating the current orchestra. If the orchestra is created by an **AudioSource** node rather than an **AudioFX** node, the value of **params** shall be 0 on every channel. See subclause 5.15.3 for more details.

Instruments may use **params** as an lvalue, that is, to assign new values to it using the **=** statement (subclause 5.8.6.6.2). In this case, when an instrument assigns to **params**, the value for the indicated controller shall be changed on the channel to which the instrument instance associated with that scope is assigned. The value of **params** is changed in all other instrument instances associated with that channel to the new value, and this change shall take effect the next time each of these instrument instances is executed at the k-rate (see subclause 5.7.3.3.6, list item 10).

5.8.7 Opcode definition

5.8.7.1 Syntactic Form

This subclause describes the definition of new opcodes. Bitstream authors may create their own opcodes according to these rules in order to encapsulate functionality and simplify instruments and the content authoring process.

<opcode definition> -> <opcode rate> **<ident> (** <formal param list> **) {**
<opcode var declarations>
<opcode statement block>
}

<opcode rate> -> **aopcode**
<opcode rate> -> **kopcode**
<opcode rate> -> **iopcode**
<opcode rate> -> **opcode**

An opcode definition has several elements. In order, they are

1. A rate tag that defines the rate at which the opcode executes, or indicates that the opcode is rate-polymorphic,
2. An identifier that defines the name of the opcode,
3. A list of zero or more formal parameters of the opcode,
4. A list of zero or more opcode variable declarations,
5. A block of statements defining the executable functionality of the opcode.

5.8.7.2 Rate tag

The rate tag describes the rate at which the opcode is to run, or else indicates that the opcode is rate-polymorphic. The four rate tags are

1. **iopcode**, indicating that the opcode runs at i-rate,

2. **kopcode**, indicating that the opcode runs at k-rate,

3. **aopcode**, indicating that the opcode runs at i-rate,

4. **opcode**, indicating that the opcode is rate-polymorphic.

See subclause 5.8.7.7 for instructions on determining the rate of a rate-polymorphic opcode.

5.8.7.3 Opcode name

Any identifier may serve as the opcode name except that the opcode name shall not be a reserved word (see subclause 5.8.8)., the name of one of the core opcodes listed in subclause 5.9, or the name of one of the core wavetable generators listed in subclause 5.10 An opcode name may be the same as the name of a variable in local or global score; there is no ambiguity so created, since the contexts in which opcode names may occur are very restricted.

No two instruments or opcodes in an orchestra shall have the same name.

5.8.7.4 Formal parameter list

5.8.7.4.1 Syntactic form

<formal param list> -> <formal param> [**,** <formal param list>]
<formal param list> -> **<NULL>**

<formal param> -> <opcode variable rate> <name>
<formal param> -> **table <ident>**

<opcode variable rate> -> **asig**
<opcode variable rate> -> **ksig**
<opcode variable rate> -> **ivar**
<opcode variable rate> -> **xsig**

<name> as defined in subclause 5.8.5.3.2.

The formal parameter list defines the calling interface to the opcode. Each formal parameter in the list has a name, a rate type, and may have an array width. If the array width is the special token **inchannels**, then the array width shall be the same as the number of input channels to the associated instrument instantiation (in the sense of subclause 5.15.2); if the array width is the special token **outchannels**, then the array width shall be the same as the number of output channels to the associated instrument instantiation (in the sense described in subclause 5.8.6.8.4).

There is no way to create user-defined opcodes with variable number of arguments in SAOL, although certain of the core opcodes have this property.

Within the opcode statement block, formal parameters may be used like any other variable. The rate tag of each formal parameter defines the rate of the variable. If an opcode is declared to be at a particular rate, then no formal parameter shall be declared faster than that rate.

There is a special rate tag **xsig** that allows formal parameters to be rate-polymorphic, see subclause 5.8.7.7.2. **xsig** shall not be the rate tag of any formal parameter unless the opcode is of type **opcode**.

5.8.7.5 Opcode variable declarations

5.8.7.5.1 Syntactic form

<opcode var declarations> -> <opcode var declaration> [<opcode var declarations>]
<opcode var declarations> -> **<NULL>**

<opcode var declaration> -> <instr variable declaration>
<opcode var declaration> -> **xsig** <namelist> **;**

<namelist> and <instr variable declaration> as defined in subclause 5.8.5.3.2 and subclause 5.8.6.5.1. In the opcode scope, the tokens **inchannels** and **outchannels** refer to the input channels and output channels of the associated instrument, respectively (in the sense described in subclause 5.8.6.8.4).

The syntax and semantics of opcode variable declarations are the same as those of instrument variable declarations as given in subclause 5.8.6.5, with the following exceptions and additions:

The opcode variable names are available only within the scope of the opcode containing them. The instrument variable declarations for the instrument instantiation associated with the opcode call are not within the scope of an opcode, and references to these names shall not be made unless the names are also explicitly declared within the opcode, in which case the variable denoted is a different one. However, standard names (subclause 5.8.6.8) are within the scope of every opcode, may be referenced within opcodes, and shall have the semantics given in subclause 5.8.6.8 as applied to the instrument instantiation associated with the opcode call. A variable shall not have a rate faster than the rate of the opcode. Standard names faster than the rate of the opcode are not defined in the opcode. The **imports** tag shall not be used for local k-rate signals when there is no global variable of the same name

The values of opcode variables are static, and are preserved from call to call referencing a particular opcode state. The values of opcode variables shall be set to 0 in an opcode state before the first call referencing that state is executed.

The **tablemap** declaration may reference any tables declared in the local scope, as well as any formal parameters that are tables.

The values of opcode variables in different states of the same opcode (due to different syntactic uses of opcode expressions, or different indexing expressions in oparray expressions) are separate and have no relationship to one another.

There is a special rate tag called **xsig** that may be used to declare opcode variables in rate-polymorphic opcodes, see subclause 5.8.7.7.2. **xsig** shall not be the rate tag of any variable unless the opcode type is **opcode**.

5.8.7.6 Opcode statement block

5.8.7.6.1 Syntactic form

<opcode statement block> -> <opcode statement> [<opcode statement block>]
<opcode statement block> -> **<NULL>**

<opcode statement> -> <statement>
<opcode statement> -> **return (** <expr list> **) ;**

<statement> as defined in subclause 5.8.6.6.1.
<expr list> as defined in subclause 5.8.6.6.

The syntax and semantics of statements in opcodes are the same as the syntax and semantics of statements in instruments, with the following exceptions and additions:

No statement in an opcode shall be faster than the rate of the opcode, as defined in subclause 5.8.7.7. A statement that is slower than the rate of the opcode shall be executed as described in subclause 5.8.7.7.3.

The assignment statement and the values of all variables refer to the opcode state associated with this particular call to this opcode, or associated with a particular indexing expression in an oparray call.

There is a special statement called **return** that is used in opcodes. This statement allows opcodes to return values back to their callers.

5.8.7.6.2 Return statement

The **return** statement allows opcodes to return values back to their callers.

The expression parameter list may contain both single-valued and array-valued expressions.

The rate of the **return** statement is the rate of the opcode containing it. No expression in the expression parameter list shall be faster than the rate of the opcode.

The **return** statement shall be evaluated as follows. Each expression in the expression parameter list is evaluated, in the order they occur in the list. The return value of the opcode is the array-value formed by sequencing the values of the expression parameters. In the case that there is only one expression parameter that is a single-valued expression, then the return value of the opcode is the single value of that expression. The return value denoted by every **return** statement within an opcode shall have the same width (although it is permissible for them to differ in the number of expressions, so long as the sum of the widths of the expressions is equal).

If an opcode has a return statements with an <expr list> of length 0, it is considered to have width 1 for the purposes of this subclause. If the opcode executes this **return** statement, the return value is undefined.

If an opcode has no **return** statements, it has a width of 1, and its return value is undefined. If no **return** statement is encountered during an opcode execution, control is returned to the calling instrument or opcode at the end of the statement block.

After a **return** statement is encountered, no further statements in the opcode are evaluated, and control returns immediately to the calling instrument or opcode.

5.8.7.7 Opcode rate

5.8.7.7.1 Introduction

This subclause describes the rules for determining the rate of a call to an opcode, and the semantics of the special tags **opcode** and **xsig.**

The rate of an opcode call depends on the type of the opcode, as follows:

1. If the opcode type is **aopcode**, calls to the opcode are a-rate.
2. If the opcode type is **kopcode**, calls to the opcode are k-rate.
3. If the opcode type is **iopcode**, calls to the opcode are i-rate.
4. If the opcode type is **opcode**, the opcode is rate-polymorphic, and the rate is as described in the next subclause.

5.8.7.7.2 Rate-polymorphic opcodes

Opcodes that are rate-polymorphic take their rates from the context in which they are called. This allows the same opcode statement block to apply to multiple calling rate contexts. Without such a construct, three versions of each opcode of this sort would have to be created and used, depending on the context.

The rate of an **opcode** opcode for a particular call is the rate of the fastest actual parameter expression (not formal parameter expression) in that call, or the rate of the fastest formal parameter in the opcode definition, or the rate of the fastest guarding **if**, **while**, or **else** expression surrounding the opcode call, or the rate of the opcode enclosing the opcode call, whichever is fastest. If an **opcode** opcode has no non-**table** parameters, and is not enclosed in a guarded block or a opcode call, then it is a **kopcode** by default.

Rate-polymorphic opcodes may contain variable declarations and formal parameter declarations using the special rate tag **xsig**. A formal parameter of type **xsig**, for a particular call to that opcode, has the same rate as the actual parameter expression in the calling expression to which it corresponds. A variable of type **xsig**, for a particular call to that opcode, has the same rate as the opcode.

Rate-polymorphic opcodes shall not contain variable declarations and statements faster than the fastest formal parameter in the opcode declaration. In particular an opcode with all **xsig** formal parameters shall not contain variable declarations other than **xsig** and **ivar** and it shall not contain statements at a defined rate faster than the initialization rate.

EXAMPLES

Given the following opcode definition:

```
opcode xop(ksig p1, xsig p2) {
  xsig v1;
  . . .
}
```

1. For the following code fragment

```
ksig k;

k = xop(1,2);
```

the rate of the opcode call is k-rate, since the formal parameter **p1** is faster than either of the actual parameters. The rate of **p2** within the call to **xop()** is i-rate, matching the actual parameter. The rate of **v1** within **xop()** is k-rate.

2. For the following code fragment

```
asig a1, a2;

a1 = xop(1,a2);
```

the rate of the opcode call is a-rate, since the actual parameter **a2** is faster than either of the formal parameters. The rates of **p2** and **v1** within the call to **xop()** are k-rate and a-rate respectively.

3. For the following code fragment

```
ksig k;
asig a;

k = xop(1,a)
```

there is a rate mismatch error, since the opcode call is a-rate, and thus shall not be assigned to a k-rate lvalue.

4. For the following code fragment

```
ksig k;
asig a1,a2;
oparray xop[10];

a1 = 0; while (a1 < 10) {
  a2 = a2 + xop[a1](1,k);
  a1 = a1 + 1;
}
```

the rate of the oparray call is a-rate, since the rate of the guarding expression is faster than any of the formal parameters or actual parameters. The rates of **p2** and **v1** within **xop()** are a-rate as well.

5.8.7.7.3 Shared variables and statements slower than the rate of the opcode

This subclause describes the rules for updating shared variables and for executing statements inside an opcode that are slower than the rate of the opcode.

In a **kopcode**, statements at the initialization rate are executed at the first call to the opcode with regard to a particular opcode state. At this call the **imports**, **exports** and **imports exports** ivars and tables are updated, as described in 5.8.6.5.3 and 5.8.6.5.4, and any wavetable generations are executed, as described in 5.8.6.5.2.

In an **aopcode**, statements at the initialization rate are executed at the first call to the opcode with regard to a particular opcode state. At this call the **imports**, **exports** and **imports exports** ivars and tables are updated, as described in 5.8.6.5.3 and 5.8.6.5.4, and any wavetable generations are executed, as described in 5.8.6.5.2. Statements at the control rate are executed at the first call of every k-cycle to the opcode with regard to a particular opcode state. At this call the **imports**, **exports** and **imports exports** ksigs and tables are updated, as described in 5.8.6.5.3 and 5.8.6.5.4. Statements that contain expressions of type **specialop** (subclause 5.9.2) are executed both at the k-rate and a-rate, and those expressions return values as specified in subclause 5.8.6.7.6 and 5.9.2.

In an **opcode**, once the rate is determined as specified in subclause 5.8.7.7.2, the same rules apply for the cases of call at **k-rate** or **a-rate** respectively.

5.8.8 Template declaration

5.8.8.1 Syntactic form

<template declaration> -> **template <** <identlist> **>** [**preset** <maplist>] **(** <identlist> **)**
map { <identlist> **} with {** <maplist> **}**
{ <instr variable declarations> <block> **}**

<maplist> -> **<** <expr list> **> ,** <maplist>
<maplist> -> **<** <expr list> **>**

<identlist> as given in subclause 5.8.5.4.
<namelist> as given in subclause 5.8.5.3.2.
<instr variable declarations> as given in subclause 5.8.6.5.1.
<expr list> as given in subclause 5.8.6.6.1.
<block> as given in subclause 5.8.6.6.1.

A template declaration allows the concise declaration of multiple instruments that are similar in processing structure and syntax, but differ in only a few key expressions or wavetable names.

5.8.8.2 Semantics

The first identifier list contains the names for the instruments declared with the template. There shall be at least one identifier in this list. The first optional maplist contains a list of the preset number lists to be associated with each instrument in the template. This maplist may be omitted, in which case there are no preset numbers associated with the template instruments. If the maplist is present, it shall contain as many lists as there are instrument names in the first identified list. The second identifier list contains the pfields for the template declaration. Each instrument declared with the template has the same list of pfields. The third identifier list contains a list of template variables that are to be replaced in the subsequent code block with expressions from the second (first required) map list. There may be no identifiers in this list, in which case each instrument declared by the template is exactly the same.

The map list takes the form of a list of lists. This list shall have as many elements as template variables declared in the third identifier list. Each sublist is a list of expressions, and shall have as many elements as instrument names in the first identifier list. The first (optional) maplist shall not contain identifiers, only numeric values.

5.8.8.3 Template instrument definitions

As many instruments are defined by the template definition as there are names in the first identifier list. To describe each of the instruments, the identifiers described in the third (template variable) list are replaced in turn by each of the expressions from the map list.

That is, to construct the code for the first instrument, the code block given is processed by replacing the first template variable with the first expression from the first map list sublist, the second template variable with the first expression from the second map list sublist, the third template variable with the first expression from the third map list sublist, and so on. To construct the code for the second instrument, the code block given is processed by replacing the first template variable with the *second* expression from the first map list sublist, the second template variable with the *second* expression from the second map list sublist, the third template variable with the second expression from the third map list sublist, and so on.

This code-block processing occurs before any other syntax-checking or rate-checking of the elements of the instruments so defined. That is, the template variables are not true signal variables, and do not need to be declared in the variable declaration block. Once the code-block processing and template expansion is complete, the resulting instruments are treated as any other instruments in the orchestra.

EXAMPLE

The following template declaration:

```
template <oneharm, threeharm> preset <3,4>,<24> (p)
  map {pitch,t,bar} with
  { <440, harm1, mysig>, <p, harm2, mysig * mysig + 2> } {

  table harm1(harm,4096,1);
  table harm3(harm,4096,3,2,1);
  asig mysig;

  mysig = oscil(t,pitch,-1);

  mysig = bar *3;
  output(mysig);
}
```

declares exactly the same two instruments as the following two instrument declarations:

```
instr oneharm(p) preset 3 4 {
  table harm1(harm,4096,1);
  table harm3(harm,4096,3,2,1);
  asig mysig;

  mysig = oscil(harm1,440,-1);

  mysig = mysig * 3;
  output(mysig);
  }

instr threeharm(p) preset 24 {
  table harm1(harm,4096,1);
  table harm3(harm,4096,3,2,1);
  asig mysig;

  mysig = oscil(harm3,p,-1);

  mysig = (mysig * mysig + 2) * 3;  // notice embedding of template expression
  output(mysig);
  }
```

5.8.9 Reserved words

The following words are reserved, and shall not be used as identifiers in a SAOL orchestra or score.

aopcode asig else exports extend global if imports inchannels instr interp iopcode ivar kopcode krate ksig map oparray opcode outbus outchannels output preset return route sasbf send sequence spatialize srate table tablemap template turnoff while with xsig

Also, variable names starting with `_sym_` are reserved for implementation-specific use (for example, bitstream detokenisation) and shall not be the name of any instrument, signal variable, wavetable, or user-defined opcode in the orchestra.

5.9 SAOL core opcode definitions and semantics

5.9.1 Introduction

This subclause describes the definitions and normative semantics for each of the core opcodes in SAOL. All core opcodes shall be implemented in every terminal that can decode Object Type 3 or 4.

For each core opcode, the following is described:

- The prototype, showing the rate of the opcode, the parameters which are required in a call to this opcode, and the rates of these parameters.
- The normative semantics of the return value. These semantics describe how to calculate the return value for each call to that opcode.
- The normative semantics of any side effects of the core opcode.

5.9.2 Specialop type

There is a special rate type for certain core opcodes called **specialop**. This rate type tag is not an actual lexical element of the SAOL language, and shall not appear in a SAOL orchestra, but is used in subsequent subclauses as a shorthand for core opcodes with these particular semantics.

Core opcodes with rate type **specialop** describe functions that map from one or more a-rate signals into a k-rate signal. That is, they have one or more parameters that vary at the a-rate, and they have normative semantics described at the a-rate, but they only return values and/or have side effects at the k-rate. When using these opcodes in expressions, they are treated as **kopcode** opcodes for the purposes of determining rate-mismatch errors (except that it is not a rate-mismatch error to pass them an a-rate signal), and as both **kopcode** and **aopcode** opcodes for the purposes of determining when to execute them.

An expression that is prohibited by the rules in subclauses 5.8.6.6 and 5.8.6.7 from being k-rate or a-rate shall not be of type **specialop**. Otherwise, an expression with one or more **specialop** components is also of type **specialop**. A **specialop** expression shall only occur in a guarded context (the code block of an **if**, or **else** statement) if the guard expression is also a **specialop** expression. A **specialop** expression shall not occur in the guarding expression or code block of the **while** statement. Calls to **specialop** type opcodes are not allowed in iopcodes, kopcodes and opcodes. If a call to a specialop type opcode is present in an aopcode, the specialop opcode is executed both at k-rate and a-rate according to rules specified in subclause 5.8.7.7.3.

The core opcodes with this type are: **fft**, **rms, sblock, downsamp**, and **decimate**.

EXAMPLE

Given the following code block:

```
asig x;
ksig y;
```

```
x = ...;
y = rms(x);          // #1
y = rms(x * 12);     // #2

if (rms(x)) {        // #3
  y = kline(...);    // #4
}

if (x) {              // #5
  y = rms(x);         // #6
}

if (rms(x) > y) {     // #7
  y = rms(x);         // #8
}
```

the following things are true. Statement #1 is a **ksig** rate statement; however, the right-side expression is a **specialop** expression and is executed at both the k-rate and at the a-rate (the assignment only occurs at the k-rate). Statement #2 is just like statement #1; the **x * 12** expression is a **specialop** expression. Statement #3 is legal, and is a **ksig** rate statement. The guard expression is a **specialop** expression and is evaluated at both the a-rate and the k-rate, and on each k-cycle on which it is nonzero, statement #4 is executed at the k-rate. Statement #5 contains a rate-mismatch error. The guard rate of the **if** statement is an a-rate expression, but statement #6 is only at the k-rate for the purposes of determining rate-mismatch errors. Statement #7 is legal. The guard expression is a **specialop** expression.

5.9.3 List of core opcodes

The several core opcodes are described in the subsequent subclauses. They are divided by category into major subclauses, but there is no normative significance in this division; it is only for clarity of presentation.

Math functions int, frac, dbamp, ampdb, abs, sgn, exp, log, sqrt, sin, cos, atan, pow, log10, asin, acos, floor, ceil, min, max

Pitch converters gettune, settune, octpch, pchoct, cpspch, pchcps, cpsoct, octcps, midipch, pchmidi, midioct, octmidi, midicps, cpsmidi

Table operations ftlen, ftloop, ftloopend, ftsr, ftbasecps, ftsetloop, ftsetend, ftsetbase, ftsetsr, tableread, tablewrite, oscil, loscil, doscil, koscil

Signal generators kline, aline, kexpon, aexpon, kphasor, aphasor, pluck, buzz, grain

Noise generators irand, krand, arand, ilinrand, klinrand, alinrand, iexprand, kexprand, aexprand, kpoissonrand, apoissonrand, igaussrand, kgaussrand, agaussrand

Filters port, hipass, lopass, bandpass, bandstop, biquad, allpass, comb, fir, iir, firt, iirt

Spectral analysis fft, ifft

Gain control rms, gain, balance, compressor

Sample conversion decimate, upsamp, downsamp, samphold, sblock

Delays delay, delay1, fracdelay

Effects reverb, chorus, flange, speedt, fx_speedc

Tempo changes gettempo, settempo

For each core opcode, an opcode prototype is given. This shows the rate of the opcode, the number of required and optional formal parameters and the rate of each of the formal parameters. Certain parameters to certain core opcodes are presented in brackets, in which case that formal parameter is optional. Certain opcodes use the "…"

notation, which means that the opcode can process an arbitrary number of parameters. The "…" is tagged with a rate for such opcodes, which is then the rate type of all of the parameters matching the varargs parameter. If there is not normative language for a particular opcode that specifies otherwise, it is a syntax error if any of the following statements apply:

- there are fewer actual parameters in the opcode call than required formal parameters
- there are more actual parameters in the opcode call than required and optional formal parameters, and the opcode definition does not include a varargs "..." section
- a particular actual parameter expression is of faster rate than the corresponding formal parameter, or than the varargs formal parameter if that is the correspondence
- a particular actual parameter expression is not single-valued, or is not table-valued when the corresponding formal parameter specifies a table.

The names associated with the formal parameters in the core opcode prototypes have no normative significance, but are used for clarity of exposition to refer to the values passed as the corresponding actual parameters when describing how to calculate the return value of the core opcode.

5.9.4 Math functions

5.9.4.1 Introduction

Each of the opcodes in this subclause computes a mathematical function. Whenever the result of calculating the function on the argument or arguments provided results in a NaN or Inf value, a run-time error shall result..

5.9.4.2 int

```
opcode int(xsig x)
```

The **int** core opcode calculates the integer part of its parameter.

The return value shall be the integer part of **x**.

5.9.4.3 frac

```
opcode frac(xsig x)
```

The **frac** core opcode calculates the fractional part of its parameter.

The return value shall be the fractional part of **x**, i.e., **x** – int(**x**). If **x** is negative, then **frac(x)** is also negative.

5.9.4.4 dbamp

```
opcode dbamp(xsig x)
```

The **dbamp** core opcode calculates the decibel equivalent of an amplitude parameter, where the amplitude of 1 corresponds to a decibel level of 90 dB. It is a run-time error if **x** is not strictly positive.

The return value shall be $90 + 20 \log_{10} \mathbf{x}$.

5.9.4.5 ampdb

```
opcode ampdb(xsig x)
```

The **ampdb** core opcode calculates the amplitude equivalent of a decibel-valued parameter, where the amplitude of 1 corresponds to a decibel level of 90 dB.

The return value shall be $10^{(\mathbf{x} - 90) / 20}$.

5.9.4.6 abs

```
opcode abs(xsig x)
```

The **abs** core opcode calculates the absolute value of a parameter.

The return value shall be **–x** if **x** < 0, or **x** otherwise.

5.9.4.7 sgn

```
opcode sgn(xsig x)
```

The **sgn** core opcode calculates the signum (sign function) of a parameter.

The return value shall be –1 if **x** < 0, 0 if **x** = 0, or 1 if **x** > 0.

5.9.4.8 exp

```
opcode exp(xsig x)
```

The **exp** core opcode calculates the exponential function.

The return value shall be $e^{\mathbf{x}}$.

5.9.4.9 log

```
opcode log(xsig x)
```

The **log** core opcode calculates the natural logarithm of a parameter.

It is a run-time error if **x** is not strictly positive.

The return value shall be log **x**.

5.9.4.10 sqrt

```
opcode sqrt(xsig x)
```

The **sqrt** core opcode calculates the square root of a parameter.

It is a run-time error if **x** is negative.

The return value shall be sqrt(**x**).

5.9.4.11 sin

```
opcode sin(xsig x)
```

The **sin** core opcode calculates the sine of a parameter given in radians.

The return value shall be sin **x**.

5.9.4.12 cos

```
opcode cos(xsig x)
```

The **cos** core opcode calculates the cosine of a parameter given in radians.

The return value shall be cos **x**.

5.9.4.13 atan

```
opcode atan(xsig x)
```

The **atan** core opcode calculates the arctangent of a parameter , in radians.

The return value shall be $\tan^{-1}$ **x**, in the range $[-\pi/2,\pi/2)$.

5.9.4.14 pow

```
opcode pow(xsig x, xsig y)
```

The **pow** core opcode calculates the to-the-power-of operation.

It shall be a run-time error if **x** is negative and **y** is not an integer.

The return value shall be $\mathbf{x}^{\mathbf{y}}$.

5.9.4.15 log10

```
opcode log10(xsig x)
```

The **log10** core opcode calculates the base-10 logarithm of a parameter.

It is a run-time error if **x** is not strictly positive.

The return value shall be $\log_{10}$ **x**.

5.9.4.16 asin

```
opcode asin(xsig x)
```

The **asin** core opcode calculates the arcsine of a parameter, in radians.

It is a run-time error if **x** is not in the range [-1, 1].

The return value shall be $\sin^{-1}$ **x**, in the range $[-\pi/2,\pi/2)$.

5.9.4.17 acos

```
opcode acos(xsig x)
```

The **acos** core opcode calculates the arccosine of a parameter, in radians.

It is a run-time error if **x** is not in the range [-1, 1].

The return value shall be $\cos^{-1}$ **x**, in the range $[0, \pi)$.

5.9.4.18 ceil

```
opcode ceil(xsig x)
```

The **ceil** core opcode calculates the ceiling of a parameter.

The return value shall be the smallest integer *y* such that **x** ≤ *y*.

5.9.4.19 floor

```
opcode floor(xsig x)
```

The **floor** core opcode calculates the floor of a parameter.

The return value shall be the greatest integer *y* such that *y* ≤ **x**.

5.9.4.20 min

```
opcode min(xsig x1[, xsig ...])
```

The **min** core opcode finds the minimum of a number of parameters.

The return value shall be the minimum value out of the parameter values.

5.9.4.21 max

```
opcode max(xsig x1[, xsig ...])
```

The **max** core opcode finds the maximum out of the parameter values.

The return value shall be the maximum value out of the parameter values.

5.9.5 Pitch converters

5.9.5.1 Introduction to pitch representations

There are four representations for pitch in a SAOL orchestra; the following twelve functions (after **gettune** and **settune**) convert them from one to another. The four representations are as follows:

- pitch-class, or **pch** representation. A pitch is represented as an integer part, which represents the octave number, where 8 shall be the octave containing middle C (C4); plus a fractional part, which represents the pitch-class, where .00 shall be C, .01 shall be C#, .02 shall be D, and so forth. Fractional parts larger than .11 (B) have no meaning in this representation; fractional parts between the pitch-class steps are rounded to the nearest pitch-class.

  For example, 7.09 is the A below middle C.

- octave-fraction, or **oct** representation. A pitch is represented as an integer part, which represents the octave number, where 8 shall be the octave containing middle C (C4); plus a fractional part, which represents a fraction of an octave, where each step of 1/12 represents a semitone.

  For example, 7.75 is the A below middle C, in equal-tempered tuning.

- MIDI pitch number representation. A pitch is represented as an integer number of semitones above or below middle C, represented as 60.

  For example, 57 is the A below middle C.

- Frequency, or **cps** representation. A pitch is represented as some number of cycles per second.

  For example, 220 Hz is the A below middle C.

Each of the pitch converters represents the conversion that is done by its name, with the new representation first and the original (parameter) representation second. Thus, **cpsmidi** is the converter that returns the frequency corresponding to a particular MIDI pitch.

Changes to the **pitch** field of an AudioBIFS **AudioSource** node (see subclause 5.15) controlling the decoding process scale the global tuning around 440 Hz just as if **settune()** was called. The value of the **pitch** field is a multiplier to be applied to 440; that is if the **pitch** field is changed to 1.1, the global tuning becomes 484 Hz. Through this mechanism, changes in the **pitch** field apply to the results of all pitch converters and the **gettune()** opcode, but to no other instructions in the orchestra.

5.9.5.2 gettune

```
opcode gettune([xsig dummy])
```

The **gettune** core opcode returns the value in Hz of the current orchestra global tuning, which is the frequency of A above middle C. The global tuning shall be set by default to 440, but can be changed using the **settune** core opcode, subclause 5.9.5.3.

The **dummy** parameter is used to specify the rate of the opcode call if desired; see subclause 5.8.7.7.2.

5.9.5.3 settune

```
kopcode settune(ksig x)
```

The **settune** core opcode sets and returns the value of the current orchestra global tuning. The global tuning is used by several pitch converters when converting between symbolic pitch representations and cycles-per-second representation.

It is a run-time error if **x** is not strictly positive. (Allowing a wide range for tuning parameters allows unusual “pitch” representations to be used).

This core opcode has side-effects, as follows: The global tuning variable shall be set to the value **x**.

The return value shall be **x**.

5.9.5.4 octpch

```
opcode octpch(xsig x)
```

The **octpch** core opcode converts pitch-class representation to octave representation, with regard to equal scale tempering.

It is a run-time error if **x** is not strictly positive.

Let the integer part of **x** be *y* and the fractional part of **x** be *z*. Then, the return value shall be calculated as follows:

z shall be “rounded” to the nearest value such that 100*z* is an integer. If $z < 0$ or $z > 0.11$, then *z* shall be set to 0 instead.

Then, the return value shall be $y + 100z / 12$.

5.9.5.5 pchoct

```
opcode pchoct(xsig x)
```

The **pchoct** core opcode converts octave representation to pitch-class representation.

It is a run-time error if **x** is not strictly positive.

Let the integer part of **x** be *y* and the fractional part of **x** be *z*. Then, the return value shall be calculated as follows:

z shall be rounded to the nearest value such that 12 *z* is an integer. Then, the return value shall be $y + 12z / 100$.

5.9.5.6 cpspch

```
opcode cpspch(xsig x)
```

The **cpspch** core opcode converts pitch-class representation to cycles-per-second representation, with regard to equal scale tempering and the global tuning.

It is a run-time error if **x** is not strictly positive.

Let the integer part of **x** be *y* and the fractional part of **x** be *z*. Then, the return value shall be calculated as follows:

z shall be "rounded" to the nearest value such that 100*z* is an integer. If $z < 0$ or $z > 11$, then *z* shall be set to 0 instead.

Further let *t* be the global tuning parameter. Then, the return value shall be $t \times 2^{(y + 100z/12 - 8.75)}$.

5.9.5.7 pchcps

```
opcode pchcps(xsig x)
```

The **pchcps** core opcode converts cycles-per-second representation to pitch-class representation, with regard to the global tuning.

It is a run-time error if **x** is not strictly positive.

The return value shall be calculated as follows.

Let *t* be the global tuning parameter. Then, let *k* be $\log_2 (\mathbf{x} / t) + 8.75$. Then, let the integer part of *k* be *y* and the fractional part of *k* be *z*, "rounded" to the nearest value such that 12*z* is an integer. The return value shall be $y + 12z / 100$.

5.9.5.8 cpsoct

```
opcode cpsoct(xsig x)
```

The **cpsoct** core opcode converts octave representation to cycles-per-second representation, with regard to the global tuning.

It is a run-time error if **x** is not strictly positive.

Let *t* be the global tuning value; then, the return value shall be $t \times 2^{(\mathbf{x} - 8.75)}$.

5.9.5.9 octcps

```
opcode octcps(xsig x)
```

The **octcps** core opcode converts cycles-per-second representation to octave representation, with regard to the global tuning.

It is a run-time error if **x** is not strictly positive.

Let *t* be the global tuning value; then, the return value shall be $\log_2$ (**x** / *t*) + 8.75.

5.9.5.10 midipch

```
opcode midipch(xsig x)
```

The **midipch** core opcode converts pitch-class representation to MIDI representation.

It is a run-time error if **x** <= 3.

Let the integer part of **x** be *y* and the fractional part of **x** be *z*. Then, the return value shall be calculated as follows:

z shall be "rounded" to the nearest value such that 100*z* is an integer. If *z* < 0 or *z* > 0.11, then *z* shall be set to 0 instead.

The return value shall be 100*z* + 12 (*y* – 3).

5.9.5.11 pchmidi

```
opcode pchmidi(xsig x)
```

The **pchmidi** core opcode converts MIDI representation to pitch-class representation.

It is a run-time error if **x** is not strictly positive.

The return value shall be calculated as follows: **x** shall be rounded to the nearest integer, then let *k* be (**x** + 36) / 12, and let *y* be the integer part of *k*, and let *z* be the fractional part of *k*. Then, the return value shall be *y* + 12*z* / 100.

5.9.5.12 midioct

```
opcode midioct(xsig x)
```

The **midioct** core opcode converts octave representation to MIDI representation.

It is a run-time error if **x** <= 3.

The return value shall be calculated as follows. Let *k* be 12 (**x** – 3). Then, the value of *k* rounded to the nearest integer shall be the return value.

5.9.5.13 octmidi

```
opcode octmidi(xsig x)
```

The **octmidi** core opcode converts MIDI representation to octave representation.

It is a run-time error if **x** is not strictly positive.

The return value shall be (**x** +36) / 12.

5.9.5.14 midicps

```
opcode midicps(xsig x)
```

The **midicps** core opcode converts cycles-per-second representation to MIDI representation, with regard to the global tuning.

It is a run-time error if **x** is not strictly positive.

Let *t* be the global tuning parameter, and let *k* be 12 $\log_2$ (**x** / *t*) + 69. Then, the return value shall be *k* rounded to the nearest nonnegative integer.

5.9.5.15 cpsmidi

```
opcode cpsmidi(xsig x)
```

The **cpsmidi** core opcode converts MIDI representation to cycles-per-second representation, with regard to the global tuning and equal scale temperament.

It is a run-time error if **x** is not strictly positive.

Let *t* be the global tuning parameter. Then, the return value shall be $t \times 2^{(\mathbf{x} - 69)/12}$.

5.9.6 Table operations

5.9.6.1 ftlen

```
opcode ftlen(table t)
```

The **ftlen** core opcode returns the length of a table. The length of a table is the value calculated based on the **size** parameter in the particular core wavetable generator as described in subclause 5.10.

The return value shall be the length of the table referenced by **t**.

5.9.6.2 ftloop

```
opcode ftloop(table t)
```

The **ftloop** core opcode returns the loop start point of a wavetable. The loop point is set either in a sound sample data block in the bitstream, or by the **ftsetloop** core opcode (see subclause 5.9.6.6), or else it is 0.

The return value shall be the loop start point (in samples) of the wavetable referenced by **t**.

5.9.6.3 ftloopend

```
opcode ftloopend(table t)
```

The **ftloopend** core opcode returns the loop end point of a wavetable. The loop point is set either in a sound sample data block in the bitstream, or by the **ftsetend** core opcode (see subclause 5.9.6.7), or else it is 0.

The return value shall be the loop end point (in samples) of the wavetable referenced by **t**.

5.9.6.4 ftsr

```
opcode ftsr(table t)
```

The **ftsr** core opcode returns the sampling rate of a wavetable. The sampling rate is set in a sound sample data block in the bitstream, or else it is 0.

The return value shall be the sampling rate, in Hz, of the wavetable referenced by **t**.

5.9.6.5 ftbasecps

```
opcode ftbasecps(table t)
```

The **ftbasecps** core opcode returns the base frequency of a wavetable, in cycles per second (Hz). The base frequency is set either in a sound sample data block in the bitstream, or in the core wavetable generator **sample** (subclause 5.10.2), or by the core opcode **ftsetbase** (subclause 5.9.6.8), or else it is 0.

The return value shall be the base frequency, in Hz, of the wavetable referenced by **t**.

5.9.6.6 ftsetloop

```
kopcode ftsetloop(table t, ksig x)
```

The **ftsetloop** core opcode sets the loop start point of a wavetable to a new value, and returns the new value.

It is a run-time error if **x** < 0, or if **x** is larger than the size of the wavetable referenced by **t**.

This core opcode has side effects, as follows: the loop start point of the wavetable **t** shall be set to sample number **x**.

The return value shall be **x**.

5.9.6.7 ftsetend

```
kopcode ftsetend(table t, ksig x)
```

The **ftsetend** core opcode sets the loop end point of a wavetable to a new value, and returns the new value. It is a run-time error if **x** < 0, or if **x** is larger than the size of the wavetable referenced by **t**.

This core opcode has side effects, as follows: the loop end point of the wavetable **t** shall be set to sample number **x**.

The return value shall be **x**.

5.9.6.8 ftsetbase

```
kopcode ftsetbase(table t, ksig x)
```

The **ftsetbase** core opcode sets the base frequency of a wavetable to a new value, and returns the new value. It is a run-time error if **x** is not strictly positive.

This core opcode has side effects, as follows: the base frequency of the wavetable **t** shall be set to **x**, where **x** is a value in Hz.

The return value shall be **x**.

5.9.6.9 ftsetsr

```
kopcode ftsetsr(table t, ksig x)
```

The **ftsetsr** core opcode sets the sampling rate parameter of a wavetable to a new value **x**, and returns the new value. It is a run-time error if **x** is not strictly positive.

This core opcode has side effects, as follows: the sampling rate of the wavetable **t** shall be set to **x**, where **x** is a value in Hz.

The return value shall be **x**.

5.9.6.10 tableread

```
opcode tableread(table t, xsig index)
```

The **tableread** core opcode returns a single value from a wavetable. It is a run-time error if **index** < 0, or if **index** is greater than or equal to (tsize-0.5), where tsize is the size of the wavetable referenced by **t**.

The return value shall be the value of the wavetable **t** at sample number **index**, where sample number 0 is the first sample in the wavetable. If **index** is not an integer, then the return value shall be interpolated from nearby points of the wavetable, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

5.9.6.11 tablewrite

```
opcode tablewrite(table t, xsig index, xsig val)
```

The **tablewrite** core opcode sets a single value in a wavetable, and returns that value. It is a run-time error if **index** < 0, or if **index** is larger than the size of the wavetable referenced by **t**.

This core opcode has side effects, as follows: **index** shall be rounded to the nearest integer, and the value of sample number **index** in the wavetable **t** shall be set to the new value **val**, where sample number 0 is the first sample in the wavetable.

The return value shall be **val**.

If global tables are written to at the a-rate by one instrument instance, it is unspecified when the new values become available to other instrument instances. In particular, it is unspecified whether the changes are available in the same orchestra cycle. The changes shall be available to all instrument instances in the next orchestra cycle, and are permitted to be available to instances of instruments sequenced later in the same orchestra cycle.

5.9.6.12 oscil

```
aopcode oscil(table t, asig freq[, ivar loops])
```

The **oscil** core opcode loops several times around the wavetable **t** at a rate of **freq** loops per second, returning values at the audio-rate. **loops** shall be rounded to the nearest integer when the opcode is evaluated. If **loops** is not provided, its value shall be –1.

It is a run-time error if **loops** is not strictly positive and is also not –1.

The return value is calculated according to the following procedure.

On the first a-rate call to **oscil** relative to a particular state, the *internal phase* shall be set to 0, and the *internal number of loops* set to **loops**. On subsequent calls, the internal phase shall be incremented by **freq/SR,** where **SR** is the orchestra sampling rate. If, after the incrementation, the internal phase is not in the interval [0,1] and the internal loop count is strictly positive, the phase shall be set to the fractional portion of its value ($p := p - \text{floor}(p)$) and the loop count decremented.

If the internal loop count is zero, the return value shall be 0. Otherwise the return value shall be the value of sample number x in the wavetable, where $x = p * l$, where p is the current internal phase, and l is the length of table **t**. If x is not an integer, then the value shall be interpolated from the nearby table values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

Note that a negative value for the frequency argument to this opcode is legal, and the phase pointer update algorithm works correctly for negative values.

NOTE - The **oscil** opcode shall not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **oscil** is referenced twice in the same a-cycle, then the effective loop frequency is twice as high as given by **freq**.

5.9.6.13 loscil

```
aopcode loscil(table t, asig freq[, ivar basefreq, ivar loopstart, ivar loopend])
```

This opcode loops around the wavetable **t**, returning values at the audio-rate. The looping continues as long as the opcode is active, and is performed at a special rate that depends on the base frequency **basefreq** and the sampling rate of the table. In this way, samples that were recorded at a particular known pitch may be interpolated to any other pitch.

If **basefreq** is not provided, it shall be set to the base frequency of the table **t** by default. If the table **t** has base frequency 0 and **basefreq** is not provided, it is a run-time error. If **basefreq** is not strictly positive, it is a runtime error. The **basefreq** parameter shall be specified in Hz.

If **loopstart** and **loopend** are not provided, they shall be set to the loop start point and loop end point of the table **t**, respectively. If **loopend** is not provided and the loop end point of **t** is 0, then it shall be set to the end of the table (*l* – 1), where *l* is the length of the table in sample points). If **loopstart** is not strictly less than **loopend**, or either is negative, it is a runtime error.

The return value is calculated according to the following procedure.

Let *l* be the length of the table, *m* be the value **loopstart** / *l*, and *n* be the value **loopend** / *l*. On the first a-rate call to **loscil** relative to a particular state, the *internal phase* shall be set to *0*. On subsequent calls, the internal phase shall be incremented by **freq * TSR / (basefreq *SR),** where **TSR** is the sampling rate of the table and **SR** is the orchestra sampling rate. If the incrementation has caused the internal phase to leave the interval [*m*,*n*] or to become less than 0, the phase shall be set to $m + p - kn$, where *p* is the internal phase and *k* is the value floor(p/n). However, if the phase pointer has not yet passed the **loopstart** value, and if an incrementation has caused the phase pointer to become less than zero, the phase pointer takes on the value zero.

The return value shall be the value of sample number *x* in the wavetable, where $x = p * l$, where *p* is the current internal phase, and *l* is the length of table **t**. If *x* is not an integer, then the value shall be interpolated from the nearby table values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

Note that a negative value for the frequency argument to this opcode is legal, and the phase pointer update algorithm works correctly for negative values.

NOTE - The **loscil** opcode shall not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **loscil** is referenced twice in the same a-cycle, then the effective loop frequency is twice as high as given by **freq**.

5.9.6.14 doscil

```
aopcode doscil(table t)
```

The **doscil** core opcode plays back a sample once, with no frequency control or looping. It is useful for sample-rate matching sampled drum sounds to an orchestra rate.

The return value is calculated according to the following procedure.

On the first a-rate call to **doscil** relative to a particular state, the *internal phase* shall be set to 0. On subsequent calls, the internal phase shall be incremented by **TSR/SR,** where **TSR** is the sampling rate of the table **t** and **SR** is the orchestra sampling rate. If, after the incrementation, the internal phase is greater than 1, then the opcode is done.

If the opcode is done, the return value shall be 0. Otherwise the return value shall be the value of sample number *x* in the wavetable, where $x = p * l$, where *p* is the current internal phase, and *l* is the length of table **t**. If *x* is not an integer, then the value shall be interpolated from the nearby table values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

NOTE - The **doscil** opcode shall not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **doscil** is referenced twice in the same a-cycle, then the sample is played back at twice its original frequency.

5.9.6.15 koscil

```
kopcode koscil(table t, ksig freq[, ivar loops])
```

This opcode loops several times around the wavetable **t** at a rate of **freq** loops per second, returning values at the control-rate. **loops** shall be rounded to the nearest integer when the opcode is evaluated. If **loops** is not provided, its value shall be set to –1.

It is a run-time error if **loops** is not strictly positive and is also not –1.

The return value is calculated according to the following procedure.

On the first k-rate call to **koscil** relative to a particular state, the *internal phase* shall be set to 0, and the *internal number of loops* set to **loops**. On subsequent calls, the internal phase shall be incremented by **freq**/**KR**, where **KR** is the orchestra control rate. If, after the incrementation, the phase is not in the interval [0,1]and the internal loop count is strictly positive, the phase shall be set to the fractional portion of its value ($p := p - \text{floor}(p)$) and the loop count decremented.

If the internal loop count is zero, the return value shall be 0. Otherwise the return value shall be the value of sample number x in the wavetable, where $x = p * l$, where p is the current internal phase, and l is the length of table **t**. If x is not an integer, then the value shall be interpolated from the nearby table values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

Note that a negative value for the frequency argument to this opcode is legal, and the phase pointer update algorithm works correctly for negative values.

NOTE - The **koscil** opcode shall not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **koscil** is referenced twice in the same k-cycle, then the effective loop frequency is twice as high as given by **freq**.

5.9.7 Signal generators

5.9.7.1 kline

```
kopcode kline(ivar x1, ivar dur1, ivar x2[, ivar dur2, ivar x3, …]);
```

The **kline** core opcode produces a line-segmented or “ramp” function, with values changing at the k-rate. This function takes **dur1** seconds to go from **x1** to **x2**, **dur2** seconds to go from **x2** to **x3**, and so on.

It is a run-time error if any of the following conditions apply:

- there are an even number of parameters
- any of the **dur** values are negative

The return value shall be calculated as follows:

On the first call to **kline** with regard to a particular state, the *internal time* shall be set to 0, the *current left point* to **x1**, the *current right point* to **x2**, and the *current duration* to **dur1**. On subsequent calls, the internal time shall be incremented by 1/**KR**, where **KR** is the orchestra control rate. So long as the internal time is thereby greater than the current duration and there is another duration parameter, the internal time shall be decremented by the current duration, the current duration shall be set to the next duration parameter, the current left point to the current right point, and the current right point to the next control point (x-value) (these steps repeat if necessary so long as the internal time is greater than the current duration). If there is no additional duration parameter, the generator is done.

If the generator is done, then the return value is 0. Otherwise, the return value is $l + (r - l)t/d$, where l is the current left point, r is the current right point, t is the internal time, and d is the current duration.

NOTE - The **kline** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **kline** is referenced twice in the same k-cycle, then the effective segment duration is half as long as given by the corresponding duration value.

5.9.7.2 aline

```
aopcode aline(ivar x1, ivar dur1, ivar x2[, ivar dur2, ivar x3, …]);
```

The **aline** core opcode produces a line-segmented or "ramp" function, with values changing at the a-rate. This function takes **dur1** seconds to go from **x1** to **x2**, **dur2** seconds to go from **x2** to **x3**, and so on.

It is a run-time error if any of the following conditions apply:

- there are an even number of parameters
- any of the **dur** values are negative

The return value shall be calculated as follows:

On the first call to **aline** with regard to a particular state, the *internal time* shall be set to 0, the *current left point* to **x1**, the *current right point* to **x2**, and the *current duration* to **dur1**. On subsequent calls, the internal time shall be incremented by 1/**SR**, where **SR** is the orchestra sampling rate. So long as the internal time is thereby greater than the current duration and there is another duration parameter, the internal time shall be decremented by the current duration, the current duration shall be set to the next duration parameter, the current left point to the current right point, and the current right point to the next control point (x-value) (these steps repeat if necessary so long as the internal time is greater than the current duration). If there is no additional duration parameter, the generator is done.

If the generator is done, then the return value is 0. Otherwise, the return value is $l + (r - l)\, t / d$, where l is the current left point, r is the current right point, t is the internal time, and d is the current duration.

NOTE - The **aline** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **aline** is referenced twice in the same a-cycle, then the effective segment duration is half as long as given by the corresponding duration value.

5.9.7.3 kexpon

```
kopcode kexpon(ivar x1, ivar dur1, ivar x2[, ivar dur2, ivar x3, …]);
```

The **kexpon** core opcode produces a segmented function made out of exponential curves, with values changing at the k-rate. This function takes **dur1** seconds to go from **x1** to **x2**, **dur2** seconds to go from **x2** to **x3**, and so on.

It is a run-time error if any of the following conditions apply:

- there are an even number of parameters
- any of the **dur** values are negative
- the **x** values are not all the same sign
- any **x** value is 0

The return value shall be calculated as follows:

On the first call to **kexpon** with regard to a particular state, the *internal time* shall be set to 0, the *current left point* to **x1**, the *current right point* to **x2**, and the *current duration* to **dur1**. On subsequent calls, the internal time shall be incremented by 1/**KR**, where **KR** is the orchestra control rate. So long as the internal time is thereby greater than the current duration and there is another duration parameter, the internal time shall be decremented by the current duration, the current duration shall be set to the next duration parameter, the current left point to the current right

point, and the current right point to the next control point (x-value) (these steps repeat if necessary so long as the internal time is greater than the current duration). If there is no additional duration parameter, the generator is done.

If the generator is done, then the return value is 0. Otherwise, the return value is $l\,(r / l)^{t/d}$, where *l* is the current left point, *r* is the current right point, *t* is the internal time, and *d* is the current duration.

NOTE - The **kexpon** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **kexpon** is referenced twice in the same k-cycle, then the effective segment duration is half as long as given by the corresponding duration value.

5.9.7.4 aexpon

```
aopcode aexpon(ivar x1, ivar dur1, ivar x2[, ivar dur2, ivar x3, …]);
```

The **aexpon** core opcode produces a segmented function made out of exponential curves, with values changing at the a-rate. This function takes **dur1** seconds to go from **x1** to **x2**, **dur2** seconds to go from **x2** to **x3**, and so on.

It is a run-time error if any of the following conditions apply:

- there are an even number of parameters
- any of the **dur** values are negative
- the **x** values are not all the same sign
- any **x** value is 0

The return value shall be calculated as follows:

On the first call to **aexpon** with regard to a particular state, the *internal time* shall be set to 0, the *current left point* to **x1**, the *current right point* to **x2**, and the *current duration* to **dur1**. On subsequent calls, the internal time shall be incremented by 1/**SR**, where **SR** is the orchestra sampling rate. . So long as the internal time is thereby greater than the current duration and there is another duration parameter, the internal time shall be decremented by the current duration, the current duration shall be set to the next duration parameter, the current left point to the current right point, and the current right point to the next control point (x-value) (these steps repeat if necessary so long as the internal time is greater than the current duration). If there is no additional duration parameter, the generator is done.

If the generator is done, then the return value is 0. Otherwise, the return value is $l\,(r / l)^{t/d}$, where *l* is the current left point, *r* is the current right point, *t* is the internal time, and *d* is the current duration.

NOTE - The **aexpon** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **aexpon** is referenced twice in the same a-cycle, then the effective segment duration is half as long as given by the corresponding duration value.

5.9.7.5 kphasor

```
kopcode kphasor(ksig cps)
```

The **kphasor** core opcode produces a moving phase value, looping from 0 to 1 repeatedly, **cps** times per second.

The return value shall be calculated as follows:

On the first call to **kphasor** with regard to a particular state, the *internal phase* shall be set to 0. On subsequent calls, the internal phase shall be incremented by **cps/KR**, where **R** is the orchestra control rate. If the internal phase is thereby not in the interval [0,1], the internal phase shall be set to the fractional part of its value (p = frac(p)). The return value is the internal phase.

Note that a negative value for the frequency argument to this opcode is legal, and the phase pointer update algorithm works correctly for negative values.

NOTE - The **kphasor** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **kphasor** is referenced twice in the same k-cycle, then the effective frequency is twice as fast as given by **cps**.

5.9.7.6 aphasor

```
aopcode aphasor(asig cps)
```

The **aphasor** opcode produces a moving phase value, looping from 0 to 1 repeatedly, **cps** times per second.

The return value shall be calculated as follows:

On the first call to **aphasor** with regard to a particular state, the *internal phase* shall be set to 0. On subsequent calls, the internal phase shall be incremented by **cps/SR**, where **SR** is the orchestra sampling rate. If the internal phase is thereby not in the interval [0,1], the internal phase shall be set to the fractional part of its value (p = frac(p)). The return value is the internal phase.

Note that a negative value for the frequency argument to this opcode is legal, and the phase pointer update algorithm works correctly for negative values.

NOTE - The **aphasor** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **aphasor** is referenced twice in the same a-cycle, then the effective frequency is twice as fast as given by **cps**.

5.9.7.7 pluck

```
aopcode pluck(asig cps, ivar buflen, table init, ksig atten, ksig smoothrate)
```

This opcode uses a simple form of the Karplus-Strong algorithm to generate plucked-string sounds by repeated sampling and smoothing of a buffer.

It is a run-time error if **buflen** is not strictly positive.

The return value is calculated as follows:

On the first call to **pluck** with regard to a particular opcode state, a *buffer* of length **buflen** shall be created and filled with the values from the table **init**, as follows. Let x be the length of the table **init**. If x is less than **buflen**, then the values of the buffer shall be set to the first **buflen** sample values of the table **init**. If x is greater than or equal to **buflen**, then the first **buflen** values of the buffer shall be set to the sample values in the table **init**, and the remainder of the buffer filled as described in this paragraph for the whole table. That is, as many full and partial cycles of the table are used as necessary to fill the buffer.

Also on the first call to **pluck** with regard to a particular state, the *internal phase* shall be set to 0, and the *smooth count* shall be set to 0.

On subsequent calls to **pluck** with regard to a state, the smooth count is incremented. If the smooth count is equal to **smoothrate**, then **smoothrate** is set to 0, and the buffer shall be smoothed, as follows. A new buffer of length **buflen** shall be created, and its values set by averaging over the current buffer. Each sample value in the new buffer shall be set to the value of the attenuated mean of the five surrounding samples of the current buffer. That is, for each sample x of the new buffer, its value shall be set to **atten** * (b[x-2] + b[x-1] + b[x] + b[x+1] + b[x+2])/5, where the b[.] notation refers to values of the current buffer, and the indices are calculated modulo **buflen** (that is, they "wrap around"). Then, the values of the current buffer shall be set to the values of the new buffer.

Whether or not the buffer has just been smoothed, the internal phase shall be incremented by **cps/SR**, where **SR** is the orchestra sampling rate, and if the resulting value is not in the interval [0,1], then the internal phase shall be set to the fractional part of the internal phase ($p = p$ – floor(p)).

The return value shall be the value of the buffer at the point *p* * **buflen**, where *p* is the internal phase. If this index is not an integer, the value shall be interpolated from nearby buffer values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

Note that a negative value for the frequency argument to this opcode is legal, and the phase pointer update algorithm works correctly for negative values.

NOTE - The **pluck** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **pluck** is referenced twice in the same a-cycle, then the effective frequency is twice as fast as given by **cps**.

5.9.7.8 buzz

```
aopcode buzz(asig cps, ksig nharm, ksig lowharm, ksig rolloff)
```

The **buzz** opcode produces a band-limited pulse train formed by adding together cosine overtones of a fundamental frequency **cps** given in Hz. These noisy sounds are useful as complex sound sources for subtractive synthesis.

lowharm gives the lowest harmonic used, where 0 is the fundamental, at frequency **cps**. It is a runtime error if **lowharm** is negative.

nharm gives the number of harmonics used starting from **lowharm**. If **nharm** is not strictly positive, then every overtone up to the orchestra Nyquist frequency is used (**nharm** shall be set to **SR** / 2 / **cps** – **lowharm**).

rolloff gives the multiplicative rolloff that defines the spectral shape. If **rolloff** is negative, then the partials alternate in phase; if |**rolloff**| > 1, then the partials increase in amplitude rather than attenuating.

The return value is calculated as follows. On the first call to **buzz** with regard to a particular scope, the *internal phase* shall be set to 0. On subsequent calls, the internal phase shall be incremented by **cps / SR**, where **SR** is the orchestra sampling rate. If, after this incrementation, the internal phase is greater than 1, the internal phase shall be set to the fractional part of its value (*p := frac(p)*).

The return value shall be

$$scale * \sum_{f=\mathbf{lowharm}}^{\mathbf{lowharm}+\mathbf{nharm}} \mathbf{rolloff}^{(f-\mathbf{lowharm})} \cos 2\pi(f+1)p$$

where *p* is the internal phase.

If |**rolloff** = 1|, **scale** = (1/(nharm+1))

else **scale** = (1-abs(**rolloff**))/(1-abs(**rolloff**$^{(\text{nharm}+1)}$)).

Note that a negative value for the frequency argument to this opcode is legal, and the phase pointer update algorithm works correctly for negative values.

5.9.7.9 grain

```
aopcode grain(table wave, table env, ksig density, ksig freq, ksig amp,
              ksig dur, ksig time, ksig phase)
```

The **grain** opcode uses granular synthesis [**GRAN**] to synthesize periodic, quasi-periodic, noisy, and textured sounds. A sound in granular synthesis is represented as the sum of a number of short sound samples or "grains" distributed throughout the time-frequency space.

wave is the waveshape for the grain. **env** is the envelope to apply to the grain. **density** is the time-spacing of grain trigger points in Hz. **freq** is the frequency in Hz at which to place each new grain. **amp** is the amplitude of

each new grain, as a scaling factor. **dur** is the duration of each grain in seconds. **time** is the offset in seconds from the trigger at which grains start (jitter). **phase** is the starting phase of each grain, in the range [0,1].

It is a run-time error if any of the following conditions apply: **density** is not positive, **dur** is negative, **time** is negative, or **phase** is not in the range [0,1].

On the first call to **grain** with regard to a particular opcode state, the *number of grains* is set to 0 and the *density clock* and *trigger clock* are both set to 0.

On the first and each subsequent a-rate call to **grain**, the following steps are executed:

The density clock is incremented by 1/SR, where SR is the orchestra sampling rate. If the density clock is thereby greater than or equal to 1/**density**, then the density clock is set to zero, and then if **time** < 1/**density**, the trigger clock is set to **time**. If **time >=** 1/**density**, the trigger clock remains at zero and no grain is created.

If the trigger clock is positive, then it is decremented by 1/SR. If the trigger clock is thereby less than or equal to zero, a new grain is dispatched. To dispatch a new grain, the number of active grains is incremented and henceforth denoted as **i**. Space shall be allocated to hold the current phase **phase[i]**, frequency **freq[i]**, amplitude **amp[i]**, duration **dur[i]**, and time **gtime[i]** of the new grain. These values shall be set to the current value of **phase**, **freq**, **amp**, **dur**, and 0 respectively.

For each active grain **i**, the current value of the grain **x[i]** is calculated as follows. There are three conditions, depending on the format of the wavetable **wave**:

1. If **wave** has both its sampling rate and base frequency parameters set, it's assumed to be a pitched sample, and is pitch-matched to **freq** in the manner of **loscil()**. Let **loopstart** be set to the loop start parameter of the table **wave**; let **loopend** be set to the loop end parameter of the table, or **l-**1 where **l** is the length of the table **wave,** if the loop end parameter is 0. Let ***m*** be the value **loopstart / l**, and let *n* be the value **loopend** / **l**. Each time after the first that the grain value is calculated, the phase **phase[i]** shall be incremented by **freq[i] * TSR / (basefreq * SR)**, where **TSR** is the table sampling rate and **basefreq** the base frequency of table **wave**. If, after this incrementation, the phase is not in the range [0,1], the phase shall be set to **phase[i] - floor(phase[i]).** The current value of the grain **x[i]** is the value of sample number **q** of the wavetable **wave**, where **q = phase[i]** * (**m-n**) + **m**. If the value of **q** is not an integer, then the value of **x[i]** shall be interpolated from the nearby table values, as described in subclause 5.8.5.2.5.

2. If **wave** has its sampling rate parameter set, but not its base frequency parameter, then it's assumed to be an unpitched sample, and is sample-rate-matched to the orchestra in the manner of **doscil()** and **freq[i]** is ignored. Each time after the first that the grain value is calculated, the phase **phase[i]** shall be incremented by **TSR/SR**, where **TSR** is the table sampling rate and **SR** is the orchestra sampling rate. If, after this incrementation, the phase is greater than 1, then the current and all future values of this grain are 0. Otherwise, the current value of the grain **x[i]** is the value of sample number **q** of the wavetable **wave**, where **q = phase[i] * l** and **l** is the length of the wavetable **wave**. If the value of **q** is not an integer, then the value of **x[i]** shall be interpolated from the nearby table values, as described in subclause 5.8.5.2.5.

3. If **wave** has neither its sampling rate nor base frequency parameters set, then it's assumed to be a waveshaped, and is oscillated in the manner of **oscil()**. Each time after the first that the grain value is **calculated**, the phase **phase[i]** shall be incremented by **freq[i]/SR**. If, after this incrementation, the phase value is outside the range [0,1], then the phase shall be set to the fractional part of its value **phase[i]-floor(phase[i])**. The current value of the grain **x[i]** shall be the value of sample number **q** of the wavetable **wave**, where **q** = **phase[i]** * **l** and **l** is the length of the wavetable **wave**. If the value of **q** is not an integer, then the value of **x[i]** shall be interpolated from the nearby table values, as described in subclause 5.8.5.2.5.

After the output value of the grain **x[i]** is calculated in one of these three manners, it is modulated by the envelope wavetable according to the grain duration. The time **gtime[i]** is incremented by **1/SR**, where **SR** is the orchestra sampling rate. If **gtime[i] > dur**, then the grain is over; it shall be noted as non-active, and its space may be deallocated. Otherwise, the modulator value **m[i]** is the value of sample number **q** of the wavetable **env**, where **q = floor(gtime[i] / dur[i] * l)** and **l** is the length of the wavetable **env**. The final output value of the grain **x[i]** is rescaled by this modulator value as in **x[i] = amp* x[i] * m[i]**.

The output value from the opcode is the sum of the output values for all active grains.

NOTE (1) - If the input values **freq, dur,** and so forth change during the use of the instrument (as they would, for example, to stochastically vary grain distribution), the values of **freq[i], dur[i]** and so forth do not change for currently-active grains. Only the parameters of new grains are affected. The parameter values of active grains do not change during the lifetime of the grain.

NOTE (2) - The **grain** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **grain** is referenced twice in the same a-cycle, than the effective frequency, duration, density, jitter, and so forth are scaled appropriately.

5.9.8 Noise generators

5.9.8.1 Note on noise generators and pseudo-random sequences

The following core opcodes generate noise, that is, pseudo-random sequences of various statistical properties. In order to provide maximum decorrelation among multiple noise generators, it is important that all references to pseudo-random generation share a single feedback state. That is, all random values required by the various states of various noise generators shall make use of sequential values from a single "master" pseudo-random sequence.

It is strictly prohibited for an implementation to maintain multiple pseudo-random sequences to draw from (using the same algorithm) for various states of noise generation opcodes, because to do so may result in strong correlations between multiple noise generators.

This point does not apply to implementations that do not use "linear congruential", "modulo feedback", or similar mathematical structures to generate pseudo-random numbers.

The standard mathematical description of probability density functions is used in this subclause. This means that if the pdf of a random variable *x* is *f*(*x*), then the probability of it taking a value in the range [*y,z*] is $\int_y^z f(x)dx$.

5.9.8.2 irand

```
iopcode irand(ivar p)
```

The **irand** core opcode generates a random number from a linear distribution.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \begin{cases} 1/2\mathbf{p} : x \in [-\mathbf{p}, \mathbf{p}] \\ 0 : otherwise \end{cases}$$

5.9.8.3 krand

```
kopcode krand(ksig p)
```

The **krand** core opcode generates random numbers from a linear distribution.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \begin{cases} 1/2\mathbf{p} : x \in [-\mathbf{p}, \mathbf{p}] \\ 0 : otherwise \end{cases}$$

5.9.8.4 arand

```
aopcode arand(asig p)
```

The **arand** core opcode generates random noise according to a linear distribution.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \begin{cases} 1/2\mathbf{p} : x \in [-\mathbf{p}, \mathbf{p}] \\ 0 : otherwise \end{cases}$$

5.9.8.5 ilinrand

```
iopcode ilinrand(ivar p1, ivar p2)
```

The **ilinrand** core opcode generates a random number from a linearly-ramped distribution.

The return value shall be a random number *x* chosen according to the pdf

p(*x*) = abs(2 / (**p2** – **p1**) * [(*x* – **p1**) / (**p2** – **p1**)]) if *x* ∈ [**p1**,**p2**]
0 otherwise

5.9.8.6 klinrand

```
kopcode klinrand(ksig p1, ksig p2)
```

The **klinrand** core opcode generates random numbers from a linearly-ramped distribution.

The return value shall be a random number *x* chosen according to the pdf

p(*x*) = abs(2 / (**p2** – **p1**) * [(*x* – **p1**) / (**p2** – **p1**)]) if *x* ∈ [**p1**,**p2**]
0 otherwise

5.9.8.7 alinrand

```
aopcode alinrand(asig p1, asig p2)
```

The **alinrand** core opcode generates random noise from a linearly-ramped distribution.

The return value shall be a random number *x* chosen according to the pdf

p(*x*) = abs(2 / (**p2** – **p1**) * [(*x* – **p1**) / (**p2** – **p1**)]) if *x* ∈ [**p1**,**p2**]
0 otherwise

5.9.8.8 iexprand

```
iopcode iexprand(ivar p1)
```

The **iexprand** core opcode generates a random number from a exponential distribution with mean **p1**. It is a run-time error if **p1** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

p(x) = 0 if *x* ≤ 0, or
k exp(-*kx*), where *k* = 1 / **p1**, otherwise.

5.9.8.9 kexprand

```
kopcode kexprand(ksig p1)
```

The **kexprand** core opcode generates random numbers from an exponential distribution with mean **p1**. It is a run-time error if **p1** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

p(x) = 0 if $x \leq 0$, or
k exp(-*kx*), where *k* = 1 / **p1**, otherwise.

5.9.8.10 aexprand

```
aopcode aexprand(asig p1)
```

The **aexprand** core opcode generates random noise according to an exponential distribution with mean **p1**. It is a run-time error if **p1** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

p(x) = 0 if $x \leq 0$, or
k exp(-*kx*), where *k* = 1 / **p1**, otherwise.

5.9.8.11 kpoissonrand

```
kopcode kpoissonrand(ksig p1)
```

The **kpoissonrand** core opcode generates a random binary (0/1) sequence of numbers such that the mean time between 1's is **p1** seconds. It is a run-time error if **p1** is not strictly positive.

On the first call to **kpoissonrand** with regard to a particular opcode state, a random number *x* shall be chosen according to the pdf

p(x) = 0 If $x \leq 0$, or
k exp(-*kx*), where *k* = 1/ (**p1 * KR)**, otherwise.

where **KR** is the orchestra control rate.

The return value shall be 0 and the floor of this random value shall be stored.

On subsequent calls, the stored value shall be decremented by 1. If the decremented value is -1, the return value shall be 1 and a new random value shall be generated and stored as described above. Otherwise, the return value shall be 0.

NOTE - The **kpoissonrand** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **kpoissonrand** is referenced twice in the same k-cycle, then the effective mean time between 1 values is half as long as given by **t**.

5.9.8.12 apoissonrand

```
aopcode apoissonrand(asig p1)
```

The **apoissonrand** core opcode generates random binary (0/1) noise such that the mean time between 1's is **p1** seconds. It is a run-time error if **p1** is not strictly positive.

On the first call to **apoissonrand** with regard to a particular opcode state, a random number *x* shall be chosen according to the pdf

p(x) = 0 If $x \leq 0$, or
k exp(-*kx*), where *k* = 1 / (**p1 * SR)**, otherwise.

where **SR** is the orchestra sampling rate.

The return value shall be 0 and the floor of this random value shall be stored.

On subsequent calls, the stored value shall be decremented by 1. If the decremented value is -1, the return value shall be 1 and a new random value shall be generated and stored as described above. Otherwise, the return value shall be 0.

NOTE - The **apoissonrand** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **apoissonrand** is referenced twice in the same a-cycle, then the effective mean time between 1 values is half as long as given by **t**.

5.9.8.13 igaussrand

```
iopcode igaussrand(ivar mean, ivar var)
```

The **igaussrand** core opcode generates a random number drawn from a Gaussian (normal) distribution with mean **mean** and variance **var**.

It is a run-time error if **var** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2\pi \times \mathbf{var}}}$$

that is, *p(x)* ~ N(**mean**, **var**) where **mean** is the mean and **var** the variance of a normal distribution.

5.9.8.14 kgaussrand

```
kopcode kgaussrand(ksig mean, ksig var)
```

The **kgaussrand** core opcode generates random numbers drawn from a Gaussian (normal) distribution with mean **mean** and variance **var**.

It is a run-time error if **var** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2\pi \times \mathbf{var}}},$$

that is, *p(x)* ~ N(**mean**, **var**) where **mean** is the mean and **var** the variance of a normal distribution.

5.9.8.15 agaussrand

```
aopcode agaussrand(asig mean, asig var)
```

The **agaussrand** core opcode generates random noise drawn from a Gaussian (normal) distribution with mean **mean** and variance **var**.

It is a run-time error if **var** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2\pi \times \mathbf{var}}},$$

that is, *p(x)* ~ N(**mean**, **var**) where **mean** is the mean and **var** the variance of a normal distribution.

5.9.9 Filters

5.9.9.1 port

```
kopcode port(ksig ctrl, ksig htime)
```

The **port** core opcode converts a step-valued control signal into a portamento signal. **ctrl** is an incoming control signal, and **htime** is the half-transition time in seconds to slide from one value to the next.

The return value is calculated as follows. On the first call to **port** with regard to a particular state, the *current value* and *old value* are both set to **ctrl**. On subsequent calls, if **ctrl** is not equal to the new value, then the old value is set to the current value, the *new value* is set to **ctrl** and the *current time* is set to 0.

If **htime** is 0, the current value is set to the new value.

The return value is calculated as follows. If the current value and new value are equal, then the return value is the new value. Otherwise, the current time is incremented by 1/**KR**, where **KR** is the orchestra control rate. Then, the current value shall be set to $o + (n - o)(1 - 2^{-t / \mathbf{htime}})$, where t is the current time, n is the new value, and o is the old value.

NOTE - The **port** opcode does not have a “proper” representation of time, but infers it from the number of calls. If the same state of **port** is referenced twice in the same k-cycle, then the effective half-transition time is half as long as given by **htime**.

5.9.9.2 hipass

```
aopcode hipass(asig input, ksig cut)
```

The **hipass** core opcode high-pass filters its input signal. **cut** is the –6 dB cut-off point of the filter, and is specified in Hz. It is a run-time error if **cut** is not strictly positive.

The particular method of high-pass filtering is not normative. Any filter with the specified characteristic may be used.

The return value shall be the result of filtering **input** with a high-pass filter at **cut**.

NOTE - The **hipass** opcode is not required to have a “proper” representation of time, but is permitted to infer it from the number of calls. If the same state of **hipass** is referenced twice in the same a-cycle, the result is unspecified.

5.9.9.3 lopass

```
aopcode lopass(asig input, ksig cut)
```

The **lopass** core opcode low-pass filters its input signal. **cut** is the –6 dB cut-off point of the filter, and is specified in Hz. It is a runtime error if **cut** is not strictly positive.

The particular method of low-pass filtering is not normative. Any filter with the specified characteristic may be used.

The return value shall be the result of filtering **input** with a low-pass filter at **cut**.

NOTE - The **lopass** opcode is not required to have a “proper” representation of time, but is permitted to infer it from the number of calls. If the same state of **lopass** is referenced twice in the same a-cycle, the result is unspecified.

5.9.9.4 bandpass

```
aopcode bandpass(asig input, ksig cf, ksig bw)
```

The **bandpass** core opcode band-pass filters its input signal. **cf** is the centre frequency of the passband, and is specified in Hz. **bw** is the bandwidth of the filter, measuring from the –6 dB cut-off point below the centre frequency to the –6 dB point above, and is specified in Hz. It is a runtime error if **cf** and **bw** are not both strictly positive.

The particular method of bandpass filtering is not normative. Any filter with the specified characteristic may be used.

The return value shall be the result of filtering **input** with a bandpass filter with centre frequency **cf** and bandwidth **bw**.

NOTE - The **bandpass** opcode is not required to have a “proper” representation of time, but is permitted to infer it from the number of calls. If the same state of **bandpass** is referenced twice in the same a-cycle, the result is unspecified.

5.9.9.5 bandstop

```
aopcode bandstop(asig input, ksig cf, ksig bw)
```

The **bandstop** core opcode band-stop (notch) filters its input signal. **cf** is the centre frequency of the stopband, and is specified in Hz. **bw** is the bandwidth of the filter, measuring from the –6 dB cut-off point below the centre frequency to the –6 dB point above, and is specified in Hz. It is a runtime error if **cf** and **bw** are not both strictly positive.

The particular method of notch filtering is not normative. Any filter with the specified characteristic may be used.

The return value shall be the result of filtering **input** with a bandstop filter with centre frequency **cf** and bandwidth **bw**.

NOTE - The **bandstop** opcode is not required to have a “proper” representation of time, but is permitted to infer it from the number of calls. If the same state of **bandstop** is referenced twice in the same a-cycle, the result is unspecified.

5.9.9.6 biquad

```
aopcode biquad(asig input, ivar b0, ivar b1, ivar b2, ivar a1, ivar a2)
```

The **biquad** core opcode performs exactly normative filtering using the canonical second-order filter in a “Transposed Direct Form II” structure. Using cascades of **biquad** opcodes allows the construction of arbitrary filters with exactly normative results.

The return value is calculated as follows. On the first call to **biquad** with regard to a particular state, the intermediate variables **ti, to, w1,** and **w2** are set to 0. Then, on the first call and each subsequent call, the following pseudo-code defines the functionality:

```
ti = input;
to = w2 + b0*ti;
w2 = w1 - a1*to + b1*ti;
w1 = -a2*to + b2*ti;
```

and the return value is **to**.

A runtime error is produced if this process produces out-of-bounds values (i.e., the filter is unstable).

The **biquad** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **biquad** is referenced twice in the same a-cycle, the effective sampling rate of the filter is twice as high as the orchestra sampling rate.

5.9.9.7 allpass

```
aopcode allpass(asig input, ivar time, ivar gain)
```

The **allpass** core opcode performs allpass filtering on an input signal. The length of the feedback delay is **time** and is specified in seconds. It is a run-time error if **time** is not strictly positive.

Let **t** be the value floor(**time** * **SR**), where **SR** is the orchestra sampling rate. On the first call to **comb** with regard to a particular state, a delay line of length **t** is initialised and set to all zeros.

On the first and each subsequent call, let **x** be the value that was inserted into the delay line **t** calls ago, or 0 if there have not been **t** calls to this state. Let the value **output** be **x – input** * **gain**. Insert the value **output** * **gain + input** into the beginning of the delay line. The return value shall be **output**.

NOTE - The **allpass** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **allpass** is referenced twice in the same a-cycle, then the effective allpass length is half as long as **len**.

5.9.9.8 comb

```
aopcode comb(asig input, ivar time, ivar gain)
```

The **comb** core opcode performs comb filtering on an input signal. The length of the feedback delay is **time** and is specified in seconds. It is a run-time error if **time** is not strictly positive.

Let **t** be the value floor(**time** * **SR**), where **SR** is the orchestra sampling rate. On the first call to **comb** with regard to a particular state, a delay line of length **t** is initialised and set to all zeros.

On the first and each subsequent call, let **x** be the value that was inserted into the delay line **t** calls ago, or 0 if there have not been **t** calls to this state. Insert the value **x** * **gain** + **input** into the beginning of the delay line. The return value shall be **x**.

NOTE - The **comb** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. That is, if the same state of **comb** is referenced twice in the same a-cycle, the effective length is half of **t**.

5.9.9.9 fir

```
aopcode fir(asig input, ksig b0[, ksig b1, ksig b2, ksig …])
```

The **fir** core opcode applies a specified FIR filter of arbitrary order to an input signal. The particular method of implementing FIR filters is not specified and left open to implementers.

The parameters **b0, b1, b2, …** specify a FIR filter

$$H(z) = \mathbf{b0} + \mathbf{b1}\, z^{-1} + \mathbf{b2}\, z^{-2} + \ldots$$

The return value shall be the successive values given by the application of this filter to the signal given by the value of **input** in successive calls to **fir**.

NOTE - The **fir** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **fir** is referenced twice in the same a-cycle, the effective filter sampling rate is double that of the orchestra sampling rate.

5.9.9.10 iir

```
aopcode iir(asig input, ksig b0[, ksig a1, ksig b1, ksig a2, ksig b2, ksig …])
```

The **iir** core opcode applies a specified IIR filter of arbitrary order to an input signal. The particular method of implementing IIR filters is not specified and left open to implementers.

The parameters **b0, b1, b2, …** and **a1, a2, …** specify an IIR filter

$$H(z) = (\mathbf{b0} + \mathbf{b1}z^{-1} + \mathbf{b2}z^{-2} + ...) / (1 + \mathbf{a1}z^{-1} + \mathbf{a2}z^{-2} + ...).$$

The return value shall be the successive values of the signal given by the application of this filter to the signal given by **input** in successive calls to **iir**. It is a run-time error if this application produces out-of-range values (that is, if the filter is unstable).

NOTE - The **iir** opcode shall not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **iir** is referenced twice in the same a-cycle, the effective filter sampling rate is double that of the orchestra sampling rate.

5.9.9.11 firt

```
aopcode firt(asig input, table t[, ksig order])
```

The **firt** core opcode applies a specified FIR filter of arbitrary order, given in a table, to an input signal. The particular method of implementing FIR filters is not specified and left open to implementers.

The values stored in samples 0, 1, 2, … **order** of table **t** specify a FIR filter

$$H(z) = \mathbf{t}[0] + \mathbf{t}[1]\,\mathbf{z}^{-1} + \mathbf{t}[2]\,\mathbf{z}^{-2} + \ldots\ \mathbf{t}[\mathbf{order}\text{-}1]\,\mathbf{z}^{-\mathbf{order}+1}$$

where array notation is used to indicate wavetable samples. If **order** is not given or is greater than the size of the wavetable **t**, then **order** shall be set to the size of the wavetable. It is a run-time error if **order** is zero or negative.

The return values shall be the successive values of the signal given by the application of this filter to the signal given by the value of **input** in successive calls to **firt**.

NOTE - The **firt** opcode shall not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **firt** is referenced twice in the same a-cycle, the effective filter sampling rate is double that of the orchestra sampling rate.

5.9.9.12 iirt

```
aopcode iirt(asig input, table a, table b[, ksig order])
```

The **iirt** core opcode applies a specified IIR filter of arbitrary order, given in two tables, to an input signal. The particular method of implementing IIR filters is not specified and left open to implementers.

The values stored in samples 1, 2, … **order** of table **a** and samples 0, 1, 2, …, **order** of wavetable **b** specify a IIR filter

$$H(z) = (\mathbf{b}[0] + \mathbf{b}[1]z^{-1} + \mathbf{b}[2]z^{-2} + ...) / (1 + \mathbf{a}[1]z^{-1} + \mathbf{a}[2]z^{-2} + ...)$$

where array notation is used to indicate wavetable samples. (Note that sample 0 of wavetable **a** is not used). If **order** is not given or is greater than the size of the larger of the two wavetables, then **order** shall be set to the size of the greater of the two wavetables. If one wavetable is smaller than given by **order**, then the “extra” values shall be taken as zero coefficients It is a run-time error if **order** is zero or negative.

The return values shall be the successive values of the signal given by the application of this filter to the signal given by the value of **input** in successive calls to **iirt**. It is a run-time error if this application produces out-of-range values (that is, if the filter is unstable).

NOTE - The **iirt** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **iirt** is referenced twice in the same a-cycle, the effective filter sampling rate is double that of the orchestra sampling rate.

5.9.10 Spectral analysis

5.9.10.1 fft

```
specialop fft(asig input, table re, table im[, ivar len, ivar shift, ivar size, table win])
```

The **fft** core opcode calculates windowed and overlapped DFT frames and places the complex-valued result in two tables. It is a "special opcode"; that is, it accepts values at the audio rate, but only returns then at the control rate.

There are several optional parameters. **len** specifies the length of the sample frame (the number of input samples to use). If **len** is zero or not provided, it is set to the next power of two greater than **SR/KR**, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate. **shift** specifies the number of samples by which to shift the analysis window. If **shift** is zero or not provided, it is set to **len**. **size** is the length of the DFT calculated by the opcode. If **size** is zero or not provided, it is set to **len**. **win** is the analysis window to apply to the analysis. If **win** is not provided, a boxcar window of length **len** is used.

It is a runtime error if any of the following apply: **len** is negative, **shift** is negative, **size** is negative, **size** is not a power of two, **size** is greater than 8192, **win** has fewer than **len** samples, **re** has fewer than **size** samples, or **im** has fewer than **size** samples.

The calculation of this opcode is as follows. On the first call to the **fft** opcode with respect to a particular state, a holding buffer of length **len** is created, and the first (**len** - **shift**) buffer samples are initialized with zeros. On each a-rate call to the opcode, the **input** sample is inserted into the next uninitialized buffer sample position. When all **len** samples in the buffer are initialized, the following steps are performed:

1. A new buffer is created of length **size**, for which each value **new[i]** is set to the value **buf[i]** * **win[i]**, where **new[i]** is the **i**th value of the new buffer, **buf[i]** is the value of the holding buffer, and **win[i]** is the value of the **i**th sample in the analysis-window wavetable. (The new buffer contains the pointwise product of the holding buffer and the analysis window). If **size > len**, then the values of **new[i]** for **i > len** are set to zero. If **size < len**, then only the first **size** values of the holding buffer are used.

2. The first **shift** samples are removed from the holding buffer and the remaining **len-shift** samples shifted to the front of the holding buffer. The **shift** samples at the end of the buffer after this shift are set to zero. If **shift** > **len**, the holding buffer is cleared.

3. The real DFT of the new buffer is calculated, resulting in a length-**size** complex vector of frequency-domain values. The real components of the DFT are placed in the first **size** samples of table **re**; the imaginary components of the DFT are placed in the first **size** samples of table **im**. The DFT is arranged such that the lowest frequencies, starting with DC, are at the zero point of the output tables, going up to the Nyquist frequency at **size/2**; the reflection of the spectrum from the Nyquist to the sampling frequency is placed in the second half of the tables.

The DFT is defined as

$$d[i] = \frac{\sum_{k=0}^{size-1} e^{-ijk\,2\pi / size} new[k]}{\sqrt{size}}$$

where **d[i]** are the resulting complex components of the DFT, 0 < **i** < **size**;
new[k] are the input samples, 0 < **k** < **size**;
and ***j*** is the square root of –1.

The return value on a particular k-cycle is 1 if a DFT was calculated since the last k-cycle, or 0 if one was not.

The **fft** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **fft** is referenced twice in the same a-cycle, then the effective FFT period is half as long as given by **len** and **shift**.

5.9.10.2 ifft

```
aopcode ifft(table re, table im[, ivar len, ivar shift, ivar size, table win])
```

The **ifft** core opcode calculates windowed and overlapped IDFTs and streams the result out as audio. **re** and **im** are wavetables that contain the real and imaginary parts of a DFT, respectively. There are several optional arguments that control the synthesis procedure. **len** is the number of output samples to use as audio; if **len** is zero or not given, it is taken as the next power of two greater than **SR/KR**, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate. **shift** is the number of samples by which the analysis window is shifted between frames; if **shift** is not given or is zero, it is taken as **len**. **size** is the size of the IDFT; if **size** is not given or is zero, it is taken as **len**. **win** is the synthesis window; if it is not given, a boxcar window of length **len** is assumed.

It is a run-time error if any of the following apply: **re** or **im** are shorter than length **size**, **win,** if given, is shorter than length **len**, **size** is not a power of two, **size** is greater than 8192, or **len**, **shift**, or **size** are negative.

The calculation for this opcode is as follows. On the first call to **ifft** with respect to a particular state, the size **size** IDFT of the tables **re** and **im** is calculated. If **re** and/or **im** are longer than **size** samples, only the first **size** samples of these tables shall be used. The result of this IDFT is a sequence of **size** values, potentially complex-valued. The real components of the first **len** elements of this sequence are multiplied point-by-point by the corresponding samples of the window **win** and placed in an output buffer of length **len**. (**out[i] = seq[i] * win[i]** for 0 < **i** < **len**).

The IDFT is calculated with the assumption that the lowest-numbered elements of the tables **re** and **im** are the lowest frequencies of the audio signal, beginning with DC in sample 0, proceeding up to the Nyquist frequency in sample **size/**2, and then the reflected spectrum in samples **size/**2 up to **size**-1.

The IDFT is defined as

$$seq[i] = \frac{\sum_{k=0}^{size-1} e^{ijk\,2\pi / size} d[k]}{\sqrt{size}}$$

where **d[i]** are the complex frequency components of the DFT, 0 < **i** < **size** (**d[i] = re[i] + *j* im[i]**)
seq[k] are the output samples, 0 < **k** < **size**;
and ***j*** is the square root of –1.

Also on the first call to **ifft** with respect to a particular state, the output point of the synthesis is set to 0.

At each call to **ifft**, the following calculation is performed. The output value of the opcode is the value of the output buffer at the output point. Then, the output point is incremented. If the output point is thereby equal to **shift**, then the following steps shall be performed:

1. The first **shift** samples of the output buffer are discarded, the remaining **len-shift** samples of the output buffer are shifted into the beginning of the buffer, and the last **shift** samples are set to 0.

2. The IDFT of the current values of the **re** and **im** wavetables is calculated as described above. The first **len** values of the real part of the resulting audio sequence are multiplied point-by-point by the synthesis window **win**,

and the result is added point-by-point to the output buffer (**out[i] = out[i] + seq[i] * win[i] * shift / len** for 0 < **i** < **len**).

3. The output point is set to 0.

NOTE - The **ifft** opcode shall not have a “proper” representation of table, but shall infer it from the number of calls. If the same state of **ifft** is referenced twice in the same a-cycle, the result is undefined.

EXAMPLE

The **ifft** and **fft** opcodes can be used together to write instruments that use FFT-based spectral modification techniques, with logical syntax at the instrument level, in which the FFT frame rate is asynchronous with the control rate. The structure of such an instrument is:

```
instr spec_mod() {
  asig out;
  ksig new_fft;
  ivar length;
  table re_original(empty, 1024);
  table im_original(empty, 1024);
  table re_modified(empty, 1024);
  table im_modified(empty, 1024);

  length = 256;
  // no windowing; place FFT in “original”
  new_fft = fft(input,re_original,im_original,1024,length);
  if (new_fft) { // read original tables, generate modified values,
                 // and place these values into modified tables.
     .
     .
     .
  }

  // and output IFFT
  out = ifft(re_modified,im_modified,1024,length);
  output(out);
  }
```

Thus, every 256 samples (assuming 256 is greater than the number of samples in the control period), we compute the 1024-point IFFT. On those k-cycles during which we compute the FFT, we modify the table values in some interesting way. The IFFT operator produces continuous output, where every 256 samples the new table data is inspected and the IFFT calculated

There is nothing preventing us from manipulating the table data every control period, but only those values present in the table at the IFFT times will actually be turned into sound.

FFTs and IFFTs do not need to be implemented in pairs; other methods (such as table calculations) can be used to generate spectra to be turned into sound with IFFT, or rudimentary audio-pattern-recognition tools can be constructed that compute functions of the FFT and return or export the results.

5.9.11 Gain control

5.9.11.1 rms

```
specialop rms(asig x[, ivar length])
```

The **rms** core opcode calculates the power in a signal. It is a “special opcode”; that is, it accepts values at the audio rate, but only returns them at the k-rate.

If **length** is not provided, it shall be set to the length of the control period. It is a run-time error if **length** is provided and is not strictly positive. The **length** parameter is specified in seconds.

The return value is calculated as follows. Let *l* be the value floor(**length * SR)**, where **SR** is the orchestra sampling rate. A buffer *b***[],** of length *l,* is maintained of the most recent values provided as the **x** parameter. Each control period, the RMS of these values is calculated as

$$p = \sqrt{\frac{\sum_{i=0}^{l-1} b[i]^2}{l}}$$

and the return value is *p*.

NOTE - The **rms** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **rms** is referenced twice in the same a-cycle, then the effective length is half as long as given by **length**.

5.9.11.2 gain

```
aopcode gain(asig x, ksig gain[, ivar length])
```

The **gain** core opcode attenuates or increases the amplitude of a signal to make its power equal to a specified power level.

If **length** is not provided, it shall be set to the length of the control period. It is a run-time error if **length** is provided and is not strictly positive. The **length** parameter is specified in seconds.

The return value is calculated as follows. Let *l* be the value floor(**length * SR)**, where **SR** is the orchestra sampling rate.

At the first call to the opcode, the attenuation level is set to 1. At each subsequent call, the input value **x** shall be stored in a buffer **b[]** of length *l.* When the buffer is full, the attenuation level is recalculated as

$$\text{gain} \Big/ \sqrt{\frac{\sum_{i=0}^{l-1} b[i]^2}{l}}$$

and the buffer is cleared.

The return value at each call is **x** * **A**, where **A** is the current attenuation level.

NOTE - The **gain** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **gain** is referenced twice in the same a-cycle, then the effective buffer length is half as long as given by **length**.

5.9.11.3 balance

```
aopcode balance(asig x, asig ref[, ivar length])
```

The **balance** core opcode attenuates or increases the amplitude of a signal to make its power equal to the power in a reference signal.

If **length** is not provided, it shall be set to the length of the control period. It is a run-time error if **length** is provided and is not strictly positive. The **length** parameter is specified in seconds.

The return value is calculated as follows. Let *l* be the value floor(**length * SR)**, where **SR**, is the orchestra sampling rate.

At the first call to the opcode, the attenuation level is set to 1. At each subsequent call, the input value **x** shall be stored in a buffer **b[]** of length *l*, and the input value **ref** stored in a buffer **r**[] of length *l*. When the buffers are full, the attenuation level is recalculated as

$$\frac{\sqrt{\frac{\sum_{i=0}^{l-1} r[i]^2}{l}}}{\sqrt{\frac{\sum_{i=0}^{l-1} b[i]^2}{l}}}$$

and the buffers are cleared.

The return value at each call is **x** * **A**, where **A** is the current attenuation level.

NOTE - The **balance** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **balance** is referenced twice in the same a-cycle, then the effective buffer length is half as long as given by **length**.

5.9.11.4 compressor

```
aopcode compressor (asig x, asig comp, ksig nfloor, ksig thresh, ksig loknee, ksig hiknee,
                    ksig ratio, ksig att, ksig rel, ivar look)
```

The **compressor** core opcode functions as an audio compressor, limiter, expander, or noise gate, using either soft-knee or hard-knee mapping, and with dynamically variable performance characteristics. It takes two audio-rate input signals, **x** and **comp**, the first of which is modified by a running analysis of the second. Both signals may be the same, or the first can be modified by a different controlling signal.

It is a run-time error if any of the following conditions apply: **thresh** < **nfloor**, **loknee < thresh**, **hiknee < loknee**, **ratio** is equal to or less than 0, **look** is negative, **att** is negative, or **rel** is negative.

compressor first examines the controlling signal **comp** by performing envelope detection. This is directed by two control values **att**, **rel**, and an initialisation value **look**, defining the attack, release, and look-ahead times (in seconds) of the envelope detector.

look is the look-ahead time (in seconds) of the envelope detector. This determines how far ahead the detector looks for a new peak in a decaying signal. If a new peak is found, the release envelope is adjusted to interpolate between the current and future peaks.

att and **rel** are the attack and release times of the envelope detector (in seconds). They define the time it takes the envelope to reach a detected peak value (for **att**) and to reach zero (for **rel**).

nfloor will set the absolute decibel floor for the system. It defines the minimum value to which the compressor is potentially reactive. For instance, if **nfloor** is 0, according to the decibel scale of SAOL, the minimum recognized absolute value for input will be $10^{-90/20}$, if **nfloor** is -10 the minimum recognized absolute value will be $10^{-(90+10)/20}$, and so on.

The envelope estimate is converted to decibels, then passed through a mapping function (see Figure 5.3) to determine what compressor action (if any) shall be taken. The mapping function is defined by four regions: the zero region, the 1:1 (no change) region, the knee, and the compression/expansion region.

The locations of these regions are defined by the decibel control values, **thresh**, **loknee**, **hiknee**, and **ratio**.

thresh is the minimum decibel level that will be allowed through. For a noise gate, this value will be greater than **nfloor**.

loknee and **hiknee** are the decibel values that define where compression or expansion will begin. These set the boundaries of a soft-knee curve joining the low-amplitude 1:1 line and the higher-amplitude compression/expansion line. If the two breakpoints are equal, a hard-knee mapping will result.

ratio, given in dB, defines compression above the knee. It defines the change in input power required for a 1dB change in output power. **ratio** values above 1 dB result in compression, with higher values yielding greater compression. Numbers between 0 and 1 result in expansion. For example, a **ratio** value of 3dB implies a compression (a 3dB change in input power produces a 1dB change in output power) while a **ratio** value of 0.5dB implies an expansion (a 0.5dB change in input power produces a 1dB change in output power).

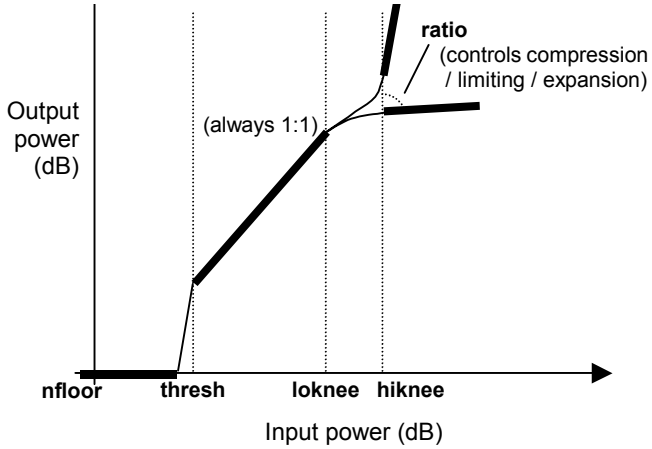


Figure 5.3 — Compressor characteristic function

The compression opcode parameters **nfloor**, **thresh**, **loknee**, and **hiknee** are defined on the decibel scale defined in subclause 5.9.4.4 for the **dbamp** core opcode. In this space, a waveform value of 1.0 maps to a decibel value of 90dB, a waveform value of 0.1 maps to 70dB, etc. Note that negative decibel values are permitted.

The actions of **compressor()** will depend on the parameter settings given. For instance, a hard-knee compressor is obtained from equal-value **loknee** and **hiknee** break-points. A noise-gate plus expander is obtained from some positive **thresh**, and a fractional **ratio** above the knee. A voice-activated music compressor (ducker) will result from feeding the music into **x** and the speech into **comp**. A voice de-esser will result from feeding the voice into both, with the **comp** version being preceded by a band-pass filter that emphasises the sibilants.

At initialisation, space shall be allocated for two buffers **xdly** and **compdly**. The length in samples of these buffers will be SR * **look**, where SR is the orchestra sampling rate. The initial values of both buffers will be set to zero.

Space is allocated for the following variables:

gain, the amplitude multiplier to be applied, initialised to 1.

change, the estimated change in envelope from sample to sample, initialised to 0.

comp1, the value of the current value of **comp** in dB, initialised to 0.

comp2, the delayed value of **comp** (from **compdelay**) in dB, initialised to zero.

env, the current envelope estimate, initialised to 0.

projEnv, the projected envelope value at the look-ahead point, initialised to 0.

At each a-rate call to **compressor()**, the following happens:

1. The sample **x** is placed into the beginning of the buffer **xdly**, pushing all values down and the oldest value off the end. The value pushed off the end is saved as **oldval**.

2. The next envelope value is calculated as follows:

   abs(**comp**) is converted to decibels. This value is called **comp1** with

   comp1 = 90 + 20*log10(abs(**comp**)).

   It is put into the end of the buffer **compdly**. The value **comp2** is taken from the beginning of the buffer **compdly**.

   if (**comp2** > **env**)

   change = (**comp2** – **env**) / (SR * **att**);

   else {

   projEnv = max(**change** * SR * **look, nfloor**)

   if ((**comp1** > **projEnv**) && (**comp2** > **comp1**))

   change = (**comp1** – **comp2**) / (SR * **rel**);

   else

   change = 0;

   }

   env = max(**env** + **change, nfloor**);

3. The amplitude multiplier **gain** is calculated as follows:

   if (**env** < **thresh**)

   gain shall first decrease monotonically from 1 to 0 and then stay at 0, moving from **thresh** towards **nfloor,** to create a noise gate. The exact input-output curve in this interval is not specified; it is left to the implementation. Implementations may also include temporal behavior in this regime to reduce onset and offset signal distortion.

   else {

   if (**env** <= **loknee**)

   gain = 1;

   else if (**env** >= **hiknee**)

   gain shall be calculated so that, above **hiknee**, an increase of **ratio** dB in input power must result in an increase of 1dB in output power. The decibel input/output curve shall be continuous at **hiknee** with either the soft or the hard knee curve.

   else

   gain shall be interpolated smoothly between the **loknee** and **hiknee** points to create a soft-knee input-output (decibel) curve. This curve shall be monotonically increasing, and have continuous

derivative equal to 1 at **loknee** and 1/**ratio** at **hiknee**. The exact input-output curve in this interval is not specified; it is left to the implementation.

}

4. The output value of the opcode shall be **oldval** multiplied by **gain**.

5.9.12 Sample conversion

5.9.12.1 decimate

```
specialop decimate(asig input)
```

The **decimate** core opcode decimates a signal from the audio rate to the control rate. It is a “special opcode”; that is, it accepts values at the audio rate, but only returns them at the k-rate.

The return value is calculated as follows. Each k-cycle, one of the values given as **input** in the preceding k-period of a-samples shall be returned.

NOTE - The **decimate** opcode is not required to have a “proper” representation of time, but is allowed to infer it from the number of calls. If the same state of **decimate** is referenced twice in the same a-cycle, then the return value for each call at the subsequent k-cycle may be taken from any of the values provided to the state during the preceding k-period.

EXAMPLE

```
oparray decimate[2];
ksig a,b,c;

a = decimate[0](1);
b = decimate[0](0);
c = decimate[1](2);
```

The value of **a** and **b** at each k-cycle shall be either 0 or 1, in an implementation-dependant manner. The value of **c** shall be 2.

5.9.12.2 upsamp

```
asig upsamp(ksig input[, table win])
```

The **upsamp** core opcode upsamples a control signal to an audio signal. **win** is an optional interpolation window. If **win** is not provided, it is taken to be a boxcar window (all values equal 1) of length **SR / KR**, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate. If **win** is provided and is shorter than **SR / KR** samples, it is zero-padded at the end to length **SR/KR** for use in this opcode (the table itself is not changed).

On the first call to **upsamp** with regard to a particular state, an output buffer of length **win** is created and set to zero. Also, the *output point* is set to 0.

On the first call to **upsamp** in a particular k-cycle with regard to a particular cycle, the output buffer is shifted by **SR / KR** samples: the first **SR / KR** samples are discarded, the remaining samples are shifted to the front of the output buffer, and the last **SR / KR** samples are set to 0. Then, the window function is scaled by **input** and added into the output buffer (**buf[i] = buf[i] + input * win[i],** 0 < **i** < length(**win**)). Then, the output point is set to 0.

On the first call and each subsequent call to **upsamp**, the return value is the value of the output buffer at the current output point. Then, the output point shall be incremented.

It is a run-time error if the same state of **upsamp** is referenced more times than the length of **win** in a single k-cycle.

5.9.12.3 downsamp

```
specialop downsamp(asig input[, table win])
```

The **downsamp** core opcode downsamples an audio signal to a control signal. It is a “special opcode”; that is, it accepts samples at the audio rate but only returns values at the control rate. **win** is an optional analysis window.

It is a run-time error if **win** is shorter than **SR / KR** samples, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate.

The return value is calculated as follows: at each k-cycle, the values of each sample of **input** provided in the previous a-cycle are placed in a buffer. If **win** is not provided, then the return value is the mean of the samples in the buffer. If **win** is provided, then the return value is calculated by multiplying the analysis window point-by-point with the input signal (**rtn** = Σ **input[i] * win[i]** for 0 < **i** < **SR/KR**, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate).

NOTE - The **decimate** opcode does not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **decimate** is referenced twice in the same a-cycle, then the return value is calculated from the input values in the second half of the k-cycle.

5.9.12.4 samphold

```
opcode samphold(xsig input, ksig gate)
```

The **samphold** core opcode gates a signal with a control signal.

The return value is calculated as follows. On the first call to **samphold** with regard to a particular state, the last passed value is set to 0. If the value of **gate** is non-zero, then the last passed value is set to **input**. The last passed value is returned.

5.9.12.5 sblock

```
specialop sblock(asig x, table t)
```

The **sblock** core opcode creates control-rate blocks of samples and places them in a wavetable. It is a “special opcode”; that is, it accepts values at the audio rate, but only returns them at the k-rate.

It is a run-time error if the table **t** is not allocated with as much space as there are samples in the control period of the orchestra.

The return value of this opcode is always 0.

This opcode has side effects, as follows. Let *k* be the number of samples in a control period. At each k-cycle, the most recent *k* values of **x** are placed in table **t** such that the oldest value is placed in sample 0.

NOTE - The **sblock** opcode shall not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **sblock** is referenced twice in the same a-cycle, then the samples placed in the table shall be the interleaved values given in the two calls during the second half of the k-period.

5.9.13 Delays

5.9.13.1 delay

```
aopcode delay(asig x, ivar t)
```

The **delay** opcode implements a fixed-length end-to-end (i.e., untapped) delay line. **t** gives the length of the delay line in seconds. It is a run-time error if **t** < 0, unless the terminal is running in a negative-time universe.

Let *y* be floor(**t** * **SR**) samples, where **SR** is the orchestra sampling rate. At each call to **delay** with respect to a particular opcode state, the value of **x** is inserted into a FIFO buffer of length *y*. The return value is the value that was inserted into the delay line *y* calls ago to **delay** with regard to the same state.

NOTE - The **delay** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **delay** is referenced twice in the same a-cycle, then the effective delay line is half as long as given by **t**.

5.9.13.2 delay1

```
aopcode delay1(asig x)
```

The **delay1** opcode implements a single-sample delay.

At each call to **delay1** with regard to a particular state, the value of **x** is stored, and the return value is the value stored on the previous call.

NOTE - The **delay1** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **delay1** is referenced twice in the same a-cycle, then the return value for the second call is the parameter value of the first.

5.9.13.3 fracdelay

```
aopcode fracdelay(ksig method[, xsig p1, xsig p2])
```

The **fracdelay** core opcode implements fractional, variable-length, and/or multitap delay lines. Several methods for manipulating the delay line are provided; in this way, **fracdelay** is like an object-oriented delay-line "class".

The semantics of **p1**and **p2** and the calculation of the return value differ depending on the value of **method**. It is a run-time error if **method** is less than 1 or greater than 5.

If **method** is 1, the "initialise" method is specified. In this case, **p1** is the length of the delay line in seconds. It is a run-time error if **p1** is not provided, or is less than 0. Any currently existing delay line in this opcode state shall be destroyed, a new delay line with the specified length (floor(**p1*** **SR**), where **SR** is the orchestra sampling rate) shall be created, and all values on this delay line shall be initialised to 0. The return value is 0. **p2** is not used, and is ignored if provided.

If **method** is 2, the "tap" method is specified. In this case, **p1** is the position of the tap in seconds. It is a run-time error if method 1 has not yet been called for this opcode state, or if **p1** is not provided, or if **p1** is less than 0, or if **p1** is greater than the most recent initialisation length. The return value is the current value of the delay line at position **p1** * **SR**, where **SR** is the orchestra sampling rate. If **p1 * SR** is not an integer, the return value shall be interpolated from the nearby values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5). **p2** is not used, and is ignored if provided.

If **method** is 3, the "set" method is specified. In this case, **p1** is the position of the insertion in seconds, and **p2** is the value to insert. It is a run-time error if method 1 has not yet been called for this opcode state, or if **p1** is not provided, or if **p1** is less than 0, or if **p1**is greater than the most recent initialisation length, or if **p2** is not provided. The value of the delay line at position floor(**p1 * SR)**, where **SR** is the orchestra sampling rate, is updated to **p2**. The return value is 0.

If **method** is 4, the "add into" method is specified. In this case, **p1** is the position of the insertion in seconds, and **p2** is the value to add in. It is a run-time error if method 1 has not yet been called for this opcode state, or if **p2** is not provided. Let *x* be the current value of the delay line at position floor(**p1 * SR**), where **SR** is the orchestra sampling rate; then, the value of the delay line at this position is updated to *x* + **p2**. The return value is *x* + **p2**.

If **method** is 5, the "shift" method is specified. It is a run-time error if method 1 has not yet been called for this opcode state. All values of the delay line are shifted forward by one sample; that is, for each sample *x* where 0 < *x* <= L, where L is the length of the delay line, the new value of sample *x* of the delay line is the current value of

sample *x* – 1. Sample 0 is set to value 0. The return value is the value shifted “off the end” of the delay line, that is the current value of sample L. **p1**and **p2** are not used, and are ignored if provided.

EXAMPLE

The following user-defined opcode implements the block diagram in Figure 5.4. We assume that the orchestra sampling rate is 10 Hz for clarity.

```
aopcode example(asig a) {
  asig t1, t2, t3, x, first;
  oparray fracdelay[1];

  if (!itime) {
    fracdelay[0](1,1);       // initialise to 1 sec long
  }

  // flow network
  fracdelay[0](3,0,a);       // insert a at beginning
  t1 = fracdelay[0](2,0);    // tap at 0
  t2 = fracdelay[0](2,0.3);  // tap at 0.3
  t3 = fracdelay[0](2,0.5);  // tap at 0.5
  fracdelay[0](4,0.1,t3);    // feedback
  fracdelay[0](4,0.8,t1+t2); // feedforward
  x = fracdelay[0](5);       // shift and get output
  return(x);
}
```

Notice the use of the oparray construction (subclause 5.8.6.7.7) to implement this network. If an oparray is not used, then each call to **fracdelay** refers to a different delay line, and the algorithm makes no sense. Also note that **fracdelay**, unlike **delay**, does not shift automatically. For “typical” operations, method 5 should be called once per a-cycle.

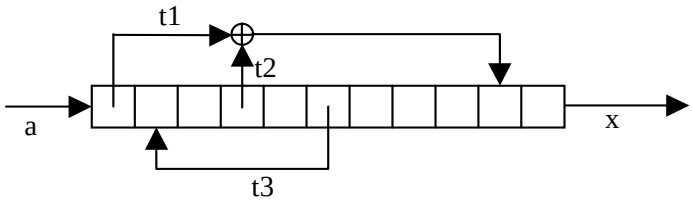


Figure 5.4 — Block diagram for ‘fracdelay’ example

5.9.14 Effects

5.9.14.1 reverb

```
aopcode reverb(asig x, ivar f0[, ivar r0, ivar f1, ivar r1, ivar …])
```

The **reverb** core opcode produces a reverberation effect according to the given parameters.

It is a run-time error if any **f** or **r** value is negative, or if there are an even number of parameters greater than 2.

If only one value **f0** is given as an argument, it is taken as a full-range reverberation time, that is, the amount of time delay until the sound amplitude is attenuated 60 dB compared to the source sound (RT60).

If more values are given, the **f** – **r** pairs represent responses at different frequencies. At each frequency **f** given as a parameter, the reverberation time (RT60) at that frequency is given by the corresponding **r** value.

The exact method of calculating the reverberation according to the specified parameters is not normative. If content authors wish to have exactly normative reverberations, they can easily be authored using the **comb**, **allpass**, **biquad**, **delay**, **fracdelay**, and other strictly normative core opcodes (q.v.).

The output shall be the reverberated sound signal.

5.9.14.2 chorus

```
asig chorus(asig x, ksig rate, ksig depth)
```

The **chorus** core opcode creates a sound with a chorusing effect, with rate **rate** and depth **depth**, from the input sound **x**. **rate** is specified in cycles per second; **depth** is specified as percent excursion with 0<=**depth**<=100. It is a run-time error if **depth**<0 or **depth**>100.

The exact method of chorusing is non-normative and left open to implementers.

5.9.14.3 flange

```
asig flange(asig x, ksig rate, ksig depth)
```

The **flange** core opcode creates a sound with a flanged effect, with rate **rate** and depth **depth**, from the input sound **x**. **rate** is specified in cycles per second; **depth** is specified as percent excursion with 0<=**depth**<=100. It is a run-time error if **depth**<0 or **depth**>100.

The exact method of flanging is non-normative and left open to implementers.

5.9.14.4 fx_speedc

```
kopcode fx_speedc(ivar speed_control_factor)
```

The **fx_speedc** core opcode creates a sound with a speed change effect. This core opcode is only available in effects-processing orchestras (see subclause 5.15.3.5); it may not be used in the SAOL block of a Structured Audio bitstream.

The **fx_speedc** core opcode directly accesses the special bus **input_bus** (at ksmps per 1/k-rate) and performs a speed change at k-rate. Then the processed samples are once stored in a special buffer defined only for the speed control and are output as the signal at a rate of *k*/**speed_control_factor**, where *k* = SR/KR, SR is the orchestra sampling rate and KR is the orchestra control rate.

This indicates that the number of the input samples and that of the output samples are different when the sampling frequency is unchanged and the **fx_speedc** core opcode shall be operated virtually at a rate of **speed_control_factor*** KR in order to keep the proper output sample-rate of *k* per control cycle. Therefore the input samples shall be also provided to the decoder at a rate of *k* per 1/ **speed_control_factor** * KR according to the **speed_control_factor** desired.

NOTE - This functionality is not supported for streaming transfer in ISO/IEC 14496-1 (MPEG-4 Systems). It is not therefore possible to apply the speed change to streaming media and the use is limited to applications such as storage media in which streaming data transfer is not required.

The exact method of speed change is non-normative and left open to implementers. Implementers are encouraged to provide the highest-quality speed change possible. As an example algorithm, the PICOLA speed change tool is described in Annex 5.D.

5.9.14.5 speedt

```
iopcode speedt(table in, table out, ivar factor)
```

The **speedt** core opcode modifies a sound sample via time-scaling.

The **speedt** core opcode fills the sound sample in the wavetable **out** with a sound derived from the wavetable **in** by time-stretching without modifying the pitch. If **factor** < 1, the sound is compressed (accelerated) by the indicated factor; if **factor** > 1, the sound is expanded (slowed down) by the indicated factor. It is a run-time error if **factor** is

not strictly positive, or if the wavetable **out** is not at least as long as **factor** multiplied by the length of the wavetable **in**.

The exact method of speed change is non-normative and left open to implementers. Implementers are encouraged to provide the highest-quality speed change possible. As an example algorithm, the PICOLA speed change tool is described in Annex 5.D.

5.9.15 Tempo functions

5.9.15.1 gettempo

```
opcode gettempo([xsig dummy])
```

The **gettempo** core opcode returns the value in beats-per-minute of the current orchestra global tempo. The tempo by default is 60 beats per minute, but can be changed through the use of the **tempo** score line (subclause 5.11.5) or the **settempo** core opcode (subclause 5.9.15.2).

The **dummy** parameter is used to specify the rate of the opcode call if desired; see subclause 5.8.7.7.2.

5.9.15.2 settempo

```
kopcode settempo(ksig x)
```

The **settempo** core opcode changes the value of the global orchestra tempo. The parameter **x** specifies the new tempo in beats-per-minute. It is a run-time error if **x** is not strictly positive. The return value is **x**.

This opcode has side-effects, as follows. All pending events are rescheduled as described in subclause 5.7.3.3.6, list item 7. The global orchestra tempo is set to **x**.

5.10 SAOL core wavetable generators

5.10.1 Introduction

This subclause describes each of the core wavetable generators in SAOL. All core wavetable generators shall be implemented in every terminal that can decode Object type 3 or 4.

For each core wavetable generator, the following is described:

- A usage description, showing the parameters that are required to be provided in a table definition utilising this core wavetable generator.
- The normative semantics of the generator. These semantics describe how to calculate values and place them in the wavetable for each table definition using this generator.

For each core wavetable generator, the first field in the table definition is the name of the generator, and the value of the expression in the second field is the size of the wavetable. Many wavetable generators also allow the value –1 in this field to signify dynamic calculation of the wavetable size. If the size is not –1, and is also not strictly greater than zero, then the syntax of the generator call is illegal. In each case, the **size** parameter shall be rounded to the nearest integer before evaluating the semantics as described below.

The subsequent expressions are the required and optional parameters to the generator. For ease of exposition, each of these parameter fields will be given a name in the description of the generators, but there is no normative significance to these names. Parameter fields enclosed in brackets are optional and may or may not occur in a table definition using that generator.

Each wavetable, as well as a block of data, has four parameters associated with it: the sampling rate loop start, loop end, and base frequency. For all wavetable generators except **sample**, these parameters shall be set to zero initially.

5.10.2 Sample

```
t1 table(sample, size, which[, skip])
```

The **sample** core wavetable generator allows the inclusion of audio samples (or other blocks of data) in the bitstream and subsequent access in the orchestra.

If **size** is –1, then the size of the table shall be the length of the audio sample. If **size** is given, and larger than the length of the audio sample, then the audio sample shall be zero-padded at the end to length **size**. If **size** is given, and smaller than the length of the audio sample, only the first **size** samples shall be used.

The **which** field identifies a sample. It is either a symbol, in which case the generator refers to a sample in the bitstream, by symbol number; or a number, in which case the generator refers to a sample stored as an **AudioBuffer** in the BIFS scene graph (ISO/IEC 14496-1 subclause 9.4.2.4).

In the case where the generator refers to a sample in the bitstream, for compliant bitstream implementations, the sample data is simply a stream of raw floating-point values. The most recent sample block of data with the given name (see subclause 5.5.2) shall be placed in the wavetable. If the bitstream sample data block contains sampling rate, loop start, loop end, and/or base frequency values, these parameters of the wavetable shall be set accordingly. If the sampling rate is not provided, it shall be set to the orchestra sampling rate by default. Any other parameters not so provided shall be set to 0.

In the case where the generator refers to a sample stored as an **AudioBuffer**, any audio coder described in ISO/IEC 14496-3 may be used to compress samples. The **children** fields of the **AudioSource** node responsible for instantiation of this orchestra refer to **AudioBuffer** nodes in this case. Each **AudioBuffer** contains, after buffering as described in ISO/IEC 14496-1 subclause 9.2.2.13, several channels of audio data. If the first child has n_0 channels, the second n_1 channels, and so forth up to child *k*-1, then this **AudioSource** node has $K = n_0 + n_1 + ... + n_{k-1}$ channels in all, and **which** shall be a value between 0 and *K-1*. Channel **which** (where **which** is rounded to the nearest integer if necessary), numbering in order across children and their channels, shall be placed in the wavetable. The sampling rate of the wavetable shall be set to the sampling rate of the **AudioBuffer** node from which channel **which** is taken. The loop start, loop end, and base frequency values shall be set to 0.

If the selected **AudioBuffer** node is not finished capturing data when the generator is executed (that is, the generator is executed less than **length** seconds after the **length** field of the **AudioBuffer** is set or changed), then the bitstream is in error. That is, this form of this generator shall only be used in cases where there is time allotted in the bitstream for the other decoders to produce samples (in real-time) before the generator executes. This is likely done by including the table generator in a score line scheduled to execute after the Composition Time (see ISO/IEC 14496-1 subclause 7.3.5) of the last audio Access Unit needed in the **AudioBuffer** node.

For standalone systems such as authoring tools, implementers are encouraged to provide access to other audio file formats and disk file access using this field (for example, to allow a filename as a string constant here). However, the only normative behaviours are those described in this subclause.

If **skip** is provided and is a positive value, it is rounded to the nearest integer, and the data placed in the wavetable begins with sample **skip**+1 of the bitstream or **AudioBuffer** sample data.

5.10.3 Data

```
t1 table(data, size, p1, p2, p3, ...)
```

The **data** core wavetable generator allows the orchestra to place data values directly into a wavetable.

If **size** is –1, then the size of the table shall be the number of data values specified. If **size** is given, and larger than the number of data values, then the wavetable shall be zero-padded at the end to length **size**. If **size** is given, and smaller than the number of data values, then only the first **size** values shall be used.

The **p1, p2, p3 ...** fields are floating-point values that shall be placed in the wavetable

5.10.4 Random

```
t1 table(random, size, dist, p1[, p2])
```

The **random** core wavetable generator fills a wavetable with pseudo-random numbers according to a given distribution. For all pseudo-random number generation algorithms, they shall be reseeded upon orchestra start-up such that each execution of an orchestra containing these instructions generates different numbers.

If **size** is –1, the generator is illegal and a run-time error generated. If the **size** field is a positive value, then this shall be the length of the table, and this many independent random numbers shall be computed to place in the table.

The **dist** field specifies which random distribution to use, and the meanings of the **p1** and **p2** fields vary accordingly.

If **dist** is 1, then a uniform distribution is used. Pseudo-random numbers are computed such that all floating-point values between **p1** and **p2** inclusive have equal probability of being chosen for any sample.

If **dist** is 2, then a linearly ramped distribution is used. Pseudo-random numbers are computed such that the probability distribution function of choosing *x* for any sample is given by

p(x) =	0	if $x \leq$ **p1** or *x* > **p2**, or
	abs(2 / (**p2** – **p1**) x [(*x* – **p1**) / (**p2** – **p1**)])	otherwise.

A run-time error is generated if **dist** is 2 and **p1 = p2**.

If **dist** is 3, then an exponential distribution is used. Pseudo-random numbers are computed such that the probability distribution function of choosing *x* for any sample is

p(x) =	0	if $x \leq 0$, or
	k exp(-*kx*), where *k* = 1 / **p1**,	otherwise.

If **dist** is 3, then **p2** is not used and is ignored if it is provided.

If **dist** is 4, then a Gaussian distribution is used. Pseudo-random numbers are computed such that the probability distribution function of choosing x for any sample is

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2\pi \times \mathbf{var}}},$$

that is, *p(x)* ~ N(**p1**, **p2**) where **p1** is the mean and **p2** the variance of a normal distribution.

If **dist** is 4, then **p2** shall be strictly greater than 0, otherwise a run-time error is generated.

If **dist** is 5, then a Poisson process is modelled, where the mean number of samples between 1's is given by an exponential distribution with mean **p1**. A pseudo-random value is computed according to *p*(*x*) as given for **dist =** 3 (the exponential distribution), above. This value is rounded to the nearest integer *y*. The first *y* values of the table (elements 0 through *y*-1) are set to 0, and the next value (element *y*) to 1. Another pseudo-random value is computed as if **dist** =3, and rounded to the nearest integer *z*. The next *z* values (elements *y* + 1 through *y* + *z* in the table) are set to 0, and the next value (element *y* + *z* + 1) to 1. This process is repeated until the table is full through element **size**. The resulting table has length **size** regardless of the values generated in the pseudo-random process; the last element may be a zero or 1.

If **dist** is 5, then **p2** is not used and is ignored if provided.

If **dist** is less than 0 or greater than 5, a run-time error is generated.

5.10.5 Step

```
t1 table(step, size, x1, y1, x2, y2, ...)
```

The **step** core wavetable generator allows arbitrary step functions to be placed in a wavetable. The step function is computed from pairs of (**x, y**) values.

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:
- **x1** is not 0,
- the x-values are not a non-decreasing sequence, or
- there are an even number of parameters, not counting the **size** parameter.

For the **step** generator, sample values 0 through **x2-**1 shall be set to **y1**, **x2** through **x3**-1 shall be set to **y2**, **x3** through **x4-1** shall be set to **y3**, and so forth.

5.10.6 Lineseg

```
t1 table(lineseg, size, x1, y1, x2, y2, ...)
```

The **lineseg** core wavetable generator allows arbitrary line-segment functions to be placed in a wavetable. The line segment function is computed from pairs of (**x**, **y**) values.

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:
- **x1** is not 0,
- the x-values are not a non-decreasing sequence, or
- there are an odd number of parameters, not counting the **size** parameter.

For the **step** generator, sample values for samples
x in the range **x1** through **x2** shall be set to **y1 +** (**y2-y1)**(*x* – **x1**) / (**x2** – **x1**)**,**
x in the range **x2** through **x3** shall be set to **y2 + (y3-y2)**(*x* **–** **x2**) / (**x3** – **x2**),

and so forth.

If any two successive x-values are equal, a discontinuous function is generated, and no values shall be calculated for the “range” corresponding to those values.

5.10.7 Expseg

```
t1 table(expseg, size, x1, y1, x2, y2, ...)
```

The **expseg** core wavetable generator allows arbitrary exponential-segment functions to be placed in a wavetable. The function is computed from pairs of (**x**, **y**) values.

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:

- **x1** is not 0,
- the x-values are not a non-decreasing sequence,
- the y-values are not all of the same sign,
- any y-value is equal to 0, or
- there are an odd number of parameters, not counting the **size** parameter.

For the **expseg** generator, sample values for samples
x in the range **x1** through **x2** shall be set to $\mathbf{y1}(\mathbf{y2}/\mathbf{y1})^{(\mathbf{x}-\mathbf{x1})(\mathbf{x2}-\mathbf{x1})}$,
x in the range **x2** through **x3** shall be set to $\mathbf{y2}(\mathbf{y3}/\mathbf{y2})^{(\mathbf{x}-\mathbf{x2})/(\mathbf{x3}-\mathbf{x2})}$,

and so forth.

If any two successive x-values are equal, a discontinuous function is generated, and no values shall be calculated for the "range" corresponding to those values.

5.10.8 Cubicseg

```
t1 table(cubicseg, size, infl1, y1, x1, y2, infl2, y3, x2, y4, infl3, y5, ...)
```

The **cubicseg** core wavetable generator creates a function made up of segments of cubic polynomials. Each segment is specified in terms of endpoints and an inflection point. If, for successive segments, the y-values at the inflection points are between the y-values at the endpoints, then the function is smooth; otherwise, the function is pointy or "comb-like".

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:

- **infl1** is not 0,
- the x-values are not a non-decreasing sequence,
- any infl-value is not strictly between the two surrounding x-values,
- there are less than two x-values, or
- the sequence of control values does not end with an y-value

For the **cubicseg** generator, sample values for samples numbered:

x in the range **infl1** to **infl2** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic polynomial that passes through (**infl1,y1**), (**x1,y2**), and (**infl2,y3**) and that has 0 derivative at **x1**;

x in the range **infl2** to **infl3** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic polynomial that passes through (**infl2,y3**), (**x2,y4**), and (**infl3,y5**) and that has 0 derivative at **x2**;

and so on.

If, for any segment, such a cubic polynomial does not exist or does not have real values through the segment range, it is a run-time error.

5.10.9 Spline

```
t1 table(spline, size, x1, y1, k2, x2, y2, k3, ...)
```

The **spline** core wavetable generator creates a smoothly varying "spline" function for a set of control points.

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:

- **x1** is not 0,
- the x-values are not a non-decreasing sequence,
- there are less than two x-values,
- there are less than 4 parameters, not including **size**, or
- the last parameter is not a *k* value

For the **spline** generator, sample values for samples numbered:

x in the range **x1** to **x2** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic polynomial that passes through (**x1,y1**), and (**x2,y2**) and that has derivative 0 at **x1** and derivative **k2** at **x2**;

x in the range **x2** to **x3** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic polynomial that passes through (**x2,y2**), and (**x3,y3**) and that has derivative **k2** at **x2** and derivative **k3** at **x3**;

x in the range **x3** to **x4** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic polynomial that passes through (**x3,y3**), and (**x4,y4**) and that has derivative **k3** at **x3** and derivative **k4** at **x4**; and so on.

The derivative of the last cubic section shall be zero at **xn**, the last x-point of the sequence.

If, for any segment, such a cubic polynomial does not exist or is not real-valued over the segment range, it is a run-time error.

5.10.10 Polynomial

```
t1 table(polynomial, size, xmin, xmax, a0, a1, a2, ...)
```

The **polynomial** core wavetable generator allows an arbitrary section of an arbitrary polynomial function to be placed in a wavetable. The polynomial function used is $p(x) = \mathbf{a0} + \mathbf{a1}x + \mathbf{a2}x^2 + ...$; it is evaluated over the range [**xmin**, **xmax**].

It is a run-time error if size is not strictly positive, or if there are not at least 3 parameters, not counting the **size** parameter, or if **xmin = xmax**.

For the **polynomial** generator, the sample value for sample *x* in the range [0,**size-1**] inclusive shall be set to

$\mathbf{a0} + \mathbf{a1}y + \mathbf{a2}y^2 + ...$, where y = **xmin** + (**size -** *x*) / **size** $\times$ (**xmax** – **xmin**).

5.10.11 Window

```
t1 table(window, size, type[, p])
```

The **window** core wavetable generator allows a windowing function to be placed in a table.

It is a run-time error if the **size** parameter is not strictly positive, or if **type** = 5 and the **p** parameter is not included.

The window type is specified by the **type** parameter. This parameter shall be rounded to the nearest integer, and then interpreted as follows:

If **type** = 1, a Hamming window shall be used. For sample number *x* in the range [0, **size** – 1], the value placed in the table shall be

$0.54 – 0.46 \cos (2\pi\ x / (\mathbf{size} – 1))$.

If **type** = 2, a Hanning (raised cosine) window shall be used. For sample number *x* in the range [0, **size** – 1], the value placed in the table shall be

0.50 (1 - cos (2π *x* / (**size** – 1))).

If **type** = 3, a Bartlett (triangular) window shall be used. For sample number *x* in the range [0, **size** – 1], the value placed in the table shall be

1 – 2 | x - (**size – 1)** / 2 | / (**size** – 1).

If **type** = 4, a Gaussian window shall be used. For sample number x in the range [0, **size**-1], the value placed in the table shall be

$e^{-c1(c2-x)(c2-x)}$, where c2 = **size**/2 and c1 = 18/(**size**)^2.

If **type** = 5, a Kaiser window shall be used, with parameter **p**. For sample number *x* in the range [0, **size** – 1], the value placed in the table shall be

$$\frac{I_0\left[p\sqrt{\left(\frac{\text{size}-1}{2}\right)^2-\left(x-\frac{\text{size}-1}{2}\right)^2}\right]}{I_0\left[p\left(\frac{\text{size}-1}{2}\right)\right]},$$

where $I_0(x)$ is the zero-order modified Bessel function of the first kind.

If **type** = 6, a boxcar window shall be used. Each sample in the range [0, **size** – 1] shall be given the value 1.

5.10.12 Harm

```
t1 table(harm, size, f1, f2, f3...)
```

The **harm** generator creates one cycle of a composite waveform made up of a weighted sum of zero-phase sinusoids.

It is a run-time error if **size** is not strictly positive.

For each sample *x* in the range [0, **size** –1], the sample shall be assigned the value

f1 sin (2 π *x*/**size**) + **f2** sin (4 π *x*/**size**) + **f3** sin (6 π *x*/**size**) + ...

5.10.13 Harm_phase

```
t1 table(harm_phase, size, f1, ph1, f2, ph2, ...)
```

The **harm_phase** core wavetable generator creates one cycle of a composite waveform made up of a weighted sum of zero-DC sinusoids, each with specified initial phase in radians.

It is a run-time error if **size** is not strictly positive, or if there are an odd number of parameters, not counting the **size** parameter.

For each sample *x* in the range [0, **size** –1], the sample shall be assigned the value

f1 sin (2 π *x*/**size** + **ph1**) + **f2** sin (4 π *x*/**size** + **ph2**) + **f3** sin (6 π *x*/**size** + **ph3**) + ...

5.10.14 Periodic

```
t1 table(periodic, size, p1, f1, ph1, p2, f2, ph2, ...)
```

The **periodic** core wavetable generator creates one cycle of an arbitrary periodic waveform, parameterised as the sum of several sinusoids with arbitrary frequency, magnitude and phase. The phase values (**ph1, ph2, ...**) are specified in radians.

It is a run-time error if **size** is not strictly positive, or if the number of parameters, not counting the **size** parameter, is not evenly divisible by three.

For each sample *x* in the range [0, **size** –1], the sample shall be assigned the value

f1 sin (2 **p1**π *x*/**size** + **ph1**) + **f2** sin (2 **p2** π *x*/**size** + **ph2**) + **f3** sin (2 **p3** π *x*/**size** + **ph3**) + ...

Any of the **p1**, **p2**, **p3**, etc. values may be zero, in which case the corresponding term of the calculation is a DC term; or non-integral, in which case there is a discontinuity at the table wrap point, or negative, which means the corresponding term evolves as a negative phase term. In all cases, the above value expression holds as specified.

5.10.15 Buzz

```
t1 table(buzz, size, nharm, lowharm, rolloff)
```

The **buzz** core wavetable generator creates one cycle of the sum of a series of spectrally-sloped cosine partials (band-limited pulse train). This waveform is a good source for subtractive synthesis.

It is a run-time error if **size** is not strictly positive, and **nharm** is also not strictly positive.

lowharm and **nharm** shall be rounded to the nearest integer before further processing.

If **size** is not strictly positive, then the size of the table is given by the highest harmonic included, such that **size** = 2 (**lowharm** + **nharm**).

If **nharm** is not strictly positive, then the number of harmonics shall be given by the size of the table, such that **nharm** is the greatest integer smaller than **size/2** – **nharm**.

For each sample *x* in the range [0, **size** –1], the sample shall be assigned the value

$$scale * \sum_{f=\mathbf{lowharm}}^{\mathbf{lowharm}+\mathbf{nharm}} \mathbf{rolloff}^{(f-\mathbf{lowharm})} \cos 2\pi fp$$

where *p* is the value *x* / **size** and ***scale*** is the value (1-abs(**rolloff**)) / (1-abs(**rolloff**$^{\mathbf{nharm}}$)).

If **rolloff** is negative, then alternating partials alternate phase direction; if **|r|** < 1, then partials attenuate as they get higher in frequency; otherwise, they stay the same or grow in magnitude; in all cases, the above value expression holds as specified.

5.10.16 Concat

```
table t1(concat, size, ft1, ft2, ...)
```

The **concat** generator allows several tables to be concatenated together into a new table.

It is a runtime error if no tables are provided as arguments.

If **size** is not strictly positive, the size of the wavetable shall be the sum of the sizes of the parameter wavetables. If **size** is strictly positive, but smaller than the sum of the sizes of the parameter wavetables, then only the first **size**

points of the parameter wavetables shall be used. If **size** is larger than the sum of the sizes of the parameter wavetables, then the generated wavetables shall be zero-padded at the end to size **size**.

The values of the wavetable shall be calculated as follows: for each sample *x* in the range [0, s_1-1], where s_1 is the size of the wavetable referenced by **p1**, the sample shall be assigned the same value as sample *x* of **p1**; for each sample *x* in the range [$\mathbf{s_1}$, s_1+s_2-1], where s_2 is the size of the wavetable referenced by **p2**, the sample shall be assigned the same value as sample $x - s_1$ of **p2**; and so on, up to sample **size**.

5.10.17 Empty

```
t1 table(empty,size)
```

The **empty** generator allocates space and fills it with zeros.

It is a run-time error if **size** is not strictly positive.

For each sample in the range [0,**size**-1], the sample is assigned value 0.

This generator is useful in conjunction with user-defined opcodes that fill up a table with data.

5.11 SASL syntax and semantics

5.11.1 Introduction

This subclause describes the syntax and semantics of the score language SASL. SASL allows the simple parametric description of events that use an orchestra to generate sound, including notes, controllers, and dynamic wavetable generation. SASL is simpler than many previously existing score languages; this is intentional, as it enables easier cross-coding of score data from other formats into SASL. Since in many cases, SASL code is automatically generated by authoring tools, it is not a great disadvantage to have relatively simple syntax and few “defaults”.

As with the SAOL description in subclause 5.8, this subclause describes a textual representation of SASL that is standardised, but stands outside of the bitstream-decoder relationship. It also describes the mapping between the textual representation and the bitstream representation. The exact normative semantics of SASL will be described in reference to the textual representation, but also apply to the tokenised bitstream representation as created via the normative tokenisation mapping.

All times in the score file (start times and durations) are specified in *score time*, which is measured in *beats*. By default, the score time is equivalent to the absolute time, and thus events with duration of one beat last one second, and an event dispatched two beats of score time after another is dispatched two seconds later by the scheduler. However, this mapping can be changed with the **tempo** command, see below.

Each score line may be prefaced by an optional ***** tag. This tag indicates that the event is a high-priority event as described in subclause 5.7.3.3.7.

NOTE - In streaming performance of Structured Audio bitstreams, some events have no timestamps. This is possible because the streaming mechanism contains intrinsic time, i.e. “right now”. For the textual score format, there is no such intrinsic time, and thus every score line in textual format is required to have a time field. A Structured Audio encoder (see Annex B) has the option of retaining or removing time fields when tokenizing the orchestra and packing tokenized score events into Access Units, depending on the requirements of the application. For the same reason, the “use if late” bitstream flag is not used in the textual score format.

5.11.2 Syntactic form

<score file> -> <score line> [<score file>]
<score file> -> <score line>

<score line> -> (*) <instr line> **<newline>**
<score line> -> (*) <control line> **<newline>**
<score line> -> (*) <tempo line> **<newline>**
<score line> -> (*) <table line> **<newline>**
<score line> -> <end line> **<newline>**

<instr line> -> [**<ident> :**] **<number> <ident> <number>** <pflist>

<control line> -> **<number>** [**<ident>**] **control <ident> <number>**

<tempo line> -> **<number> tempo <number>**

<table line> -> **<number> table <ident> <ident>** <pflist>

<end line> -> **<number> end**

<pflist> -> **<number>** [<pflist>]
<pflist> -> **<NULL>**

<number> as given in subclause 5.8.2.3.

<ident> as given in subclause 5.8.2.2.

5.11.3 Instr line

The **instr** line specifies the construction of an instrument instantiation at a given time.

The first identifier, if given, is a label that is used to identify the instantiation for use with further control events.

The first number is the score time of the event. As much precision as desired may be used to specify times; however, instruments are only dispatched as fast as the orchestra control rate, as described in subclause 5.7.3.3. Event times do not have to be received, or present in the score file, in temporal order.

The second identifier (the first required identifier) is the name of the instrument, used to select one instrument from the orchestra described in the SAOL bitstream element. It is a syntax error if there is not an instrument with this name in the orchestra when the orchestra is started.

The second number is the score duration of the instrument instance. When the instrument instantiation is created, a termination event shall be scheduled (see subclause 5.7.3.3) at the time given by the sum of the instantiation time and the note duration. If this field is –1, then the instrument shall have no scheduled duration.

The pflist is the list of parameter fields to be passed to the instrument instance when it is created. If there are more pfields specified in the instrument declaration than elements of this list, the remaining pfields shall be set to 0 upon instantiation. If there are fewer pfields than elements, the extra elements shall be ignored.

5.11.4 Control line

The **control** line specifies a control instruction to be passed to the orchestra, or to a set of running instruments.

The first number is the score time of the control event. When this time arrives in the orchestra, the control event is dispatched according to its particular semantics.

The first identifier, if provided, is a label specifying which instrument instances are to receive the event. If this label is provided, when the control event is dispatched, any active instrument instances that were created by **instr** events with the same label receive the control event. If the label is provided, and there are no such active instrument instances, the control event shall be ignored. If the label is not provided, then the control event references a global variable of the orchestra.

The second identifier (the first required identifier) is the name of a variable that will receive the event. For labelled control lines, the name references a variable in instruments that were created based upon **instr** events with the same label. If there is no such name in a particular instrument instance, then the control event shall be ignored for that instance. For unlabelled lines, the name references a global variable of the orchestra with the same name. If there is no such global variable, then the control event shall be ignored.

The second number is the new value for the control variable. When the control event is dispatched, variables in the orchestra as identified in the preceding paragraph shall have their values set to this value.

5.11.5 Tempo line

The **tempo** line in the score specifies the new tempo for the decoding process. The tempo is specified in beats-per-minute; the default tempo shall be sixty beats per minute, and thus by default the score time is measured in seconds.

The first number in the tempo line is the score time at which the tempo changes. When this time arrives, the tempo event shall be dispatched as described in subclause 5.7.3.3, list item 7.

The second number is the new tempo, specified in beats per minute. Consequently, one beat lasts 60/**tempo** seconds, so that a tempo of 120 beats per minute is twice as fast as the default. When a tempo line is decoded, the time numbers in the score continue progressively, with the increments now in accordance with the new time unit.

5.11.6 Table line

The **table** line in the score specifies the creation or destruction of a wavetable.

The first number in the score line is the score time at which the wavetable is created or destroyed. For creation events, the wavetable shall be created at this time. For destruction events, the wavetable shall not be destroyed before this time.

The first identifier is the name of the wavetable. This name references a wavetable in the global orchestra scope.

The second identifier is either the name of the table generator, or the special name **destroy**. It is a syntax error if this identifier is not the name of one of the core wavetable generators listed in subclause 5.10, or the special name **destroy**.

The pfield list is the list of parameters to the particular core wavetable generator. Not every sequence of parameters is legal for every table generator; see the definitions in subclause 5.10.

The **sample** core wavetable generator refers to a sound sample (see subclause 5.10.2). Implementations providing textual interfaces are suggested to provide access to commonly-used “soundfile” formats in the first pfield as a string constant. However, this is non-normative; the only normative aspect is as follows. In a bitstream **table** score line object, the **refers_to_sample** bit may be set. If this is the case, then the **sample** token of that score line object shall refer to another bitstream object containing the sample data, and it is this sample data that shall be placed in the wavetable.

When the dispatch time of the table event is received, if the table line references the **destroy** name, then any global wavetable with that name may be destroyed and its memory freed. If the table line specifies creation of a wavetable, and there is already a global wavetable with the same name, the new wavetable replaces the existing wavetable. That is, the global wavetable with that name may be destroyed and its memory freed.

When a new table is to be created, memory space is allocated for the table and filled with data according to the particular wavetable generator. Any reference to a wavetable with this name (including indirect references through import into a instrument instance) in existing or new instrument instances shall be taken as direction to the new wavetable.

NOTE - According to this paragraph, the wavetables referenced by running instrument instances shall be replaced upon dispatch of a **table** score line using the same name. That is, in the midst of the sound generation process, when the **table** score line is dispatched, any table-reference opcodes in an instrument referencing that name shift reference to the new wavetable.

5.11.7 End line

The **end** line in the score specifies the end of the sound-generation process. The number given is the end time, in score time, for the orchestra. When this time is reached, the orchestra ceases, and all future Composition Buffers based on this Structured Audio decoding process contain only 0 values.

5.12 SAOL/SASL tokenisation

5.12.1 Introduction

This subclause describes the normative process of mapping between the SAOL textual format used to describe syntax and semantics in subclause 5.8, and the tokenised bitstream representation used in the bitstream definition in subclause 5.5. The textual representation stands outside of the bitstream-decoder relationship, and as such is not required to be implemented or used. The only aspect of SAOL decoding that is strictly normative is the process of turning a tokenised bitstream representation into sound as described in subclause 5.7. However, it is highly recommended that implementations that allow access to bitstream contents use the textual representation described in subclause 5.8 rather than the tokenised representation. It is nearly impossible for a human reader to understand a SAOL program presented in tokenised format.

5.12.2 SAOL tokenisation

To tokenise a textual SAOL orchestra, the following steps shall be performed. First, the orchestra shall be divided into lexical elements, where a lexical element is one of the following:

1. A punctuation mark,
2. A reserved word (see subclause 5.8.9),
3. A standard name (see subclause 5.8.6.8),
4. A core opcode name (see subclause 5.9.3),
5. A core wavetable generator name (see subclause 5.10),
6. A symbolic constant (a string, integer, or floating-point constant; see subclause 5.8.2.3), or
7. An identifier (see subclause 5.8.2.2).

Whitespace (see subclause 5.8.2.6) may be used to separate lexical elements as desired; in some cases, it is required in order to lexically disambiguate the orchestra. In neither case shall whitespace be treated as a lexical element of the orchestra. Comments (see subclause 5.8.2.5) may be used in the textual SAOL orchestra but are removed upon lexical analysis; comments are not preserved through a tokenisation/detokenisation sequence.

After lexical analysis, all identifiers in the orchestra shall be numbered with symbol values, so that a single symbol is associated with a particular textual identifier. All identifiers that are textually equivalent (equal under string comparison) shall be associated with the same symbol regardless of their syntactic scope. This association of symbols to identifiers is called the *symbol table.*

Using the lexical analysis and the symbol table, a tokenised representation of the orchestra may be produced. The lexical analysis is scanned in the order it was presented in the textual representation, and for each lexical element:

- If the element is of type (1) – (5) from above, the token value associated in the table in Annex 5.A with that element shall be produced.

- If the element is of type (6) from above, one of the special tokens 0xF1, 0xF2, 0xF3, 0xF4 shall be produced, depending on the type of the symbolic constant, and the succeeding bitstream element shall be the bitstream representation of the value. For integer constants in the range [0,255], either token 0xF1 or token 0xF4 may be produced.

- If the element is of type 7, the special token 0xF0 shall be produced, and the succeeding bitstream element shall be the symbol associated with the identifier in the symbol table.

After the sequence of lexical elements presented in the textual orchestra is tokenised, the special token 0xFF, representing end-of-orchestra, shall be produced.

5.12.3 SASL tokenisation

A SASL score shall be tokenised with respect to a particular SAOL orchestra, since the symbol values must correspond in order for the semantics to be according to the author's intent.

To tokenise a SASL file, the following steps are taken. First, the SASL file is divided into lexical elements, where each element is either an identifier, a reserved word, the name of a core wavetable generator, or a number. After lexical analysis, each identifier shall be associated with the appropriate symbol number from the SAOL orchestra reference. That is, for the associated SAOL orchestra, if there is an identifier in the orchestra equivalent to the identifier in the score, the identifier in the score shall receive the same symbol number that it received in the orchestra. If there is no such identifier in the orchestra, any unused symbol number may be assigned to the identifier in the score.

Using the lexical analysis and the symbol table, a tokenised representation of the orchestra may be produced. Each score line is taken in turn, in the order presented in the textual representation, and used to produce a **score_line** bitstream element, according to the semantics in subclause 5.11 and the bitstream syntax for the various score elements, as given in subclause 5.5.2.

5.13 Sample Bank syntax and semantics

5.13.1 Introduction

This subclause describes the operation of the Sample Bank synthesis method for Object types 2 and 4. In Object type 2, only Sample Bank and MIDI class types shall appear in the bitstream, and this subclause describes the normative process of generating sound from a Sample Bank bitstream data element and a sequence of MIDI instructions. In Object type 4, Sample Banks are used in the context of a SAOL instrument as described in subclause 5.8.6.7.15, and this subclause describes the normative process of generating sound and returning it to the SAOL decoding process, depending on the Sample Bank bitstream data element and the particular call to **sasbf**.

The Structured Audio Sample Bank Format is derived from the MIDI Manufacturers Association (MMA) Downloadable Sounds format, which has been adopted as a standard for sample-data exchange by electronic musical instrument and PC audio manufacturers. The MMA DLS-2 standard [**DLS2**] contains provisions which, through reference, constitute provisions of this part of this standard. Subsequent amendments or revision to this publication do not apply, but parties are encouraged to investigate the possibility of applying the most recent editions of the referenced document. In particular, though it is assumed that typically an MPEG encoder will strip all unnecessary chunks from valid DLS files, DLS "chunks" that are not defined in the DLS Level 2 text shall be syntactically valid within an MPEG Sample Bank stream, though it is acceptable to ignore them and not use them for synthesis if not specifically defined in the DLS-2 standard.

5.13.2 Elements of bitstream

The **SASBF** bitstream element is a block of data defined by the MIDI DLS file structure [**DLS**]. This block of data is opaque to the MPEG-4 bitstream parser; among other reasons for this opaqueness, it contains values that are byte-swapped (big-endian) compared to the rest of the MPEG-4 bitstream.

5.13.3 Decoding process

5.13.3.1 Object type 2

5.13.3.1.1 Overview

In Object type 2, all synthesis is performed through wavetable-bank synthesis as defined in the MIDI DLS Level 2 specification [**DLS2**]. Control is through the use of the Standard MIDIFile bitstream element and the MIDI command bitstream element.

5.13.3.1.2 Channels, sample format, and sampling rate

For the purposes of attaching a Object type 2 Structured Audio (i.e., SASBF) decoder to an AudioBIFS **AudioSource** node, the resulting audio stream shall have two channels, 32-bit floating point samples, and a 22050 Hz sampling rate. Calculation is not required to occur in stereo 32-bit samples; if the internal representation is otherwise, the result shall be converted to this format after decoding is complete.

5.13.3.1.3 Decoder configuration

In the stream header (decoder configuration element), one or more **sbf** chunks may appear; a standard MIDIFile **midi_file** chunks may also appear. The **sbf** chunks are passed to the SASBF synthesiser, which uses the data there to prepare for synthesis as described in [**DLS2**]. The **midi_file** chunk, if any, is unrolled and organised in time according to its semantics as given in [**MIDI**].

5.13.3.1.4 Runtime decoding

Two types of events may control runtime synthesis in Object type 2: cached MIDI events that were transmitted as a MIDI file in the stream header, and real-time MIDI events that are transmitted over the streaming connection.

A *decoding clock* is maintained to control dispatch of events, but the exact properties of this clock are nonnormative. At each step, the MIDI scheduler shall dispatch any MIDI events that have arrived in the bitstream with Decoding Time Stamp less than the current value of the decoding clock, as well as any MIDI events sequenced in **midi_file** chunks in the stream header whose unrolled time-stamps are less than the current value of the decoding clock.

Interactive manipulations to the **speed** field of the **AudioSource** scene graph node pointing to this decoding process affect the playback speed of cached Standard MIDIFile events, but have no effect on the dispatch of streaming MIDI events. Thus, if might be the case that events that are synchronised between a MIDIFile and a streaming control end up no longer synchronised if the **speed** field is manipulated.

The sound that results shall be the sound described by the synthesis process in [**DLS2**] according to the sample banks in the stream header and the sequence of MIDI bytes dispatched by the scheduler. These sound samples are provided to the **AudioSource** node in the scene graph that references this bitstream as the output of the Object type 2 Structured Audio decoder.

NOTE - For compliant operation, the sound output from the SASBF synthesis process is not immediately turned into audio and played to the listener. Instead, the sound is made available for further processing by the AudioBIFS scene graph. Implementations that make use of DLS-2 hardware synthesisers that produce analogue output shall “recapture” this output and convert back to a digital signal for use in the scene graph. This is necessary because interactive scene-graph manipulations may alter, attenuate, or eliminate the sound produced in this synthesis process before it is finally played to the listener.

5.13.3.2 Object type 4

5.13.3.2.1 Overview

In Object type 4, the SASBF synthesiser is not controlled directly by MIDI data, but dispatched note-by-note in response to commands in SAOL. The **sasbf** statement (subclause 5.8.6.7.15) performs this dispatching function.

5.13.3.2.2 Decoder configuration

In Object type 4 operation, **sbf** data chunks in the bitstream configuration header are passed to the SASBF synthesiser, where they are used to prepare it for real-time synthesis.

5.13.3.2.3 Runtime decoding

In Object type 4 operation, each note of synthesis is performed separately. The note-on command is executed when the i-rate pass of an instrument containing the **sasbf** expression is executed. This instruction contains note, velocity, preset, and bank select values; the synthesis of one note indicated by this preset number and bank is performed for this note number and velocity according to the SASBF instrument. The resulting stereo sound is returned by the **sasbf** expression (see subclause 5.8.6.7.15).

The SASBF decoder shall make use of the MIDI controller and other continuously changing MIDI information for the channel specified. This data is not passed directly into the SASBF synthesiser in the **sasbf** command, but is made available in an implementation-specific manner. See subclause 5.8.6.7.15.

5.14 MIDI semantics

5.14.1 Introduction

This subclause describes the normative decoding process for Object type 1 implementations, and the normative mapping from MIDI events in the stream information header and bitstream data into SAOL semantics for Object type 3 and 4 implementations.

The MIDI standards referenced are standardised externally by the MIDI Manufacturers Association. In particular, we reference the Standard MIDI File format, the MIDI protocol, and the General MIDI patch mapping, all standardised in **[MIDI]**. The MIDI terminology used in this subclause is defined in that document.

5.14.2 Object type 1 decoding process

Little normative needs be said about the Object type 1 decoding process. The rules given in **[MIDI]** apply as standardised in those documents. As described in subclause 5.6, only **midi** and **midi_file** bitstream elements shall occur in a Object type 1 bitstream.

There are no normative aspects to producing sound in Object type 1.

5.14.3 Mapping MIDI events into orchestra control

5.14.3.1 Introduction

For Object types 3 and 4, events coded as MIDI data shall be converted, when they are received in the terminal as part of a Standard MIDI File or MIDI event, into the appropriate scheduler semantics. This subclause lists the various MIDI events and their corresponding semantics in MPEG-4. These semantics apply only to Object types 3 and 4, not to Object types 1 and 2. For the latter, the semantics of MIDI events are exactly as given in [MIDI].

5.14.3.2 MIDI events

5.14.3.2.1 Introduction

This subclause describes the semantics of the various types of events that may arrive in a continuous bitstream as a **MIDI_event** object. The syntax of these objects is standardised externally in **[MIDI]**.

5.14.3.2.2 Extended channel values

An actual MIDI Channel event in **[MIDI]** has a channel number in the range 0…15. There is no direct way in **[MIDI]** to specify a channel number outside this range. Each MIDI input port, output port or track chunk is associated with a distinct stream or collection of MIDI events and a corresponding distinct set of 16 channels (some of which may be unused). MIDI applications commonly use port names, track names or other labels to identify different channel sets.

Extended channel numbers are used in MPEG-4 to avoid the need for such channel set labels. In MPEG-4, the **channel** value of a **MIDI_event** is not limited to the range 0…15. Instead, an extended channel value is generated based on both the original MIDI channel number and a number associated with the port or stream that is the source of the event. Subclause 5.14.3.3.4 describes the mapping used with Standard MIDI Files. Annex 5.F.2 describes a recommended mapping that may be used with events from a live MIDI device.

5.14.3.2.3 NoteOn

```
noteon channel note velocity
```

When a **noteon** event is received with nonzero velocity, the instrument in the orchestra (if any) currently assigned to channel **channel** shall be instantiated with duration –1 and the first two p-fields set to **note** and **velocity**. Each value of **MIDIctrl[]** within the instrument instance is set to the most recent value of a controller change for that controller on channel **channel** or to the default value (see subclause 5.14.3.3.2) if there have been no controller changes for that controller on that channel. The value of **MIDIbend** is set to the most recent value of the MIDI pitch bend. The value of **MIDItouch** is set to the most recent aftertouch value on the channel.

If there is no instrument currently assigned to channel **channel**, there is no action associated with this event.

An instrument instance created in response to a **noteon** message on a particular channel is referred to as being “on” that channel.

noteon messages with velocity 0 shall be treated as **noteoff** messages, see subclause 5.14.3.2.4.

5.14.3.2.4 NoteOff

```
noteoff channel note velocity
```

When a **noteoff** event is received, each instrument instance on channel **channel** that was instantiated with note number **note** is scheduled for termination at the end of the k-cycle; that is, its **released** flag is set, and if the instrument does not call **extend**, it shall be de-instantiated after the current k-cycle of computation.

If **MIDIctrl[64]** on the indicated channel is non-zero, then the execution of the **noteoff** event shall be delayed until **MIDIctrl[64]** on the indicated channel becomes zero. This behaviour maintains whether the value of **MIDIctrl[64]** is set in the bitstream or by assignment to the **MIDIctrl** standard name (see subclause 5.8.6.8.9).

5.14.3.2.5 Control change

```
cc channel controller value
```

When a **cc** or control change event is received, the new value of the specified controller is set to **value**. This value shall be cached so that future instrument instances on the given channel have access to it; also, all currently active instrument instances on the channel **channel** shall have the standard name **MIDIctrl[controller]** updated to **value**.

5.14.3.2.6 Aftertouch

```
touch channel note velocity
```

When a **touch** event is received, the value of the **MIDItouch** variable of each instrument instance on channel **channel** that was instantiated with note number **note** is set to **velocity**.

5.14.3.2.7 Channel aftertouch

```
ctouch channel velocity
```

When a **ctouch** event is received, the value of the **MIDItouch** variable of each instrument instance on channel **channel** is set to **velocity**.

5.14.3.2.8 Program change

```
pchange channel program
```

When a **pchange** event is received, the current instrument receiving events on channel **channel** shall be changed to the instrument with preset number **program** (see subclause 5.8.6.4). Only the instrument with preset number **program** is assigned to the channel. If there is no instrument with this preset number, then future note-on events on the channel, until another program change is received, shall be ignored.

5.14.3.2.9 Bank select

```
bankselect channel bank
```

When a **bankselect** event is received, the next time a **pchange** event is received, the current instrument receiving events on channel **channel** shall be changed to the instrument with preset number **bank** * 128 + **program**. The **bankselect** event has no direct effect by itself; it only changes the meaning of future **pchange** events on the channel.

5.14.3.2.10 Pitch wheel change

```
pwheel channel value
```

When a **pwheel** event is received, the **MIDIbend** value for each instrument instance on channel **channel** shall be set to **value**.

5.14.3.2.11 All notes off

```
notesoff
```

When a **notesoff** event is received, all instrument instances in the orchestra created by a MIDI NoteOn events (subclause 5.14.3.2.3) are scheduled for termination at the end of the current k-cycle; that is, the released flag is set, and if the instrument does not call extend, it shall be de-instantiated after the current k-cycle of computation.

If the **MIDIctrl**[64] value for an instance is non-zero, then the execution of the termination shall be delayed until the **MIDIctrl**[64] value becomes zero.

These semantics correspond to the behavioral intentions of the MIDI All Notes Off command: the release cycle of notes are permitted to decay naturally, and sustain pedal semantics are obeyed. The **MIDIctrl**[123] value, normally 0, shall be set to 1 for all associated MIDIctrl accesses during the orchestra cycle the All Notes Off Command is processed, so that dynamic instruments spawned from the MIDI instrument may detect the All Notes Off command.

5.14.3.2.12 Tempo change

```
tempochange value
```

When a **tempochange** event is received, the global orchestra tempo is changed as described in subclause 5.7.3.3.6, list item 7. **value** here indicates a beats/minute value as in the SASL **tempo** score event (subclause 5.11.5); it shall be converted from the native MIDI tempo format (see [MIDI]) to this format.

5.14.3.2.13 All sound off

```
soundoff
```

When a **soundoff** event is received, all instrument instances in the orchestra created by MIDI NoteOn events (subclause 5.14.3.2.3) are terminated at the end of the current k-cycle. Instruments may not save themselves from termination using the extend statement in this case, and the status of the **MIDIctrl**[64] is irrelevant. The **MIDIctrl**[120] value, normally 0, shall be set to 1 for all associated **MIDIctrl** accesses during the orchestra cycle the All Sound Off Command is processed, so that dynamic instruments spawned from the MIDI instrument may detect the All Sound Off command.

These semantics correspond to the behavioral intentions of the MIDI All Sound Off command: an immediate ending of all sound making.

5.14.3.2.14 MIDI messages not respected

The following MIDI messages have no meaning in MPEG-4 Object types 3 and 4:

Local Control
Omni Mode On/Off
Mono Mode On/Off
Poly Mode On/Off
System Exclusive
Tune Request
Timing Clock
Song Select/Continue/Stop
Song Position
Active Sensing
Reset

5.14.3.2.15 The MIDI Master Channel

All instances created by SASL instr statements and SAOL send statements, and any dynamic instruments instantiated from instances created by SASL instr statements and SAOL send statements, see the status of the MIDI Master Channel in the values of the **MIDIctrl**[], **MIDIbend**, **MIDIwheel**, and **channel**, and **preset** standard names. For these instances, the value of the **channel** standard name reflects the extended channel number of the MIDI master channel, and the value of the **preset** standard name reflects the value of the last pchange event on the MIDI Master Channel.The values of the **MIDIctrl**[], **MIDIbend**, **MIDIwheel** are identical to the values seen in an instance instantiated via a MIDI NoteOn event from the MIDI Master Channel.

The identity of the MIDI Master Channel is determined as follows:

If a MIDI source is directly connected to the orchestra, as described in Annex 5.F, then the MIDI Master Channel is MIDI channel 0 of this source.

If no MIDI source is directly connected to the orchestra, if an SA_access_unit MIDI source is present, the MIDI Master Channel is channel 0 of this source.

If no MIDI source is directly connected to the orchestra, and if no SA_access_unit MIDI source is present, the MIDI Master Channel is channel 0 of the first MIDI File track that contains a NoteOff, NoteOn, CChange, PChange, Pwheel, Touch, or CTouch command on channel 0.

Decoder implementations may provide a way to specify the MIDI Master Channel directly, superseding these rules, if the decoder permits the direct connection of MIDI sources to the orchestra.

5.14.3.3 Standard MIDI Files

5.14.3.3.1 Introduction

MIDI files have data with the same semantics as the MIDI messages described above; however, the timing semantics are more complicated due to the use of multiple tracks and delta-time timestamps.

5.14.3.3.2 Overview of MIDI file processing

To process a **midi_file** stream information element, the following steps shall be taken. First, the entire stream element is parsed and cached. Then, using the sequence instructions and the sequencer model described in **[MIDI]**, the delta-times of the various **midi_file** events are converted into score event times (in beats). During this step, the actual channel numbers of these events are also mapped to extended **channel** values, as described in subclause 5.14.3.3.4. Converting delta-times to score event times requires converting each **midi_file** track chunk into a timelist containing **midi_event** objects and then interleaving the various track timelists.

5.14.3.3.3 Converting MIDI file track chunks

To convert the track chunk into a timelist, first parse the track chunk to generate a series of MIDI events. This requires converting MIDI file delta-times to MIDI event score times relative to the beginning of each track chunk. It is also necessary to map **midi_file** event channel numbers to extended **midi_event channel** values, as described in subclause 5.14.3.3.4. When a Set Tempo **midi_file** meta-event is processed to generate a corresponding tempo change **midi_event**, the tempo value shall be converted from the microseconds-per-quarter-note units used in [MIDI] to the beats-per-minute units used in MPEG-4. If the MIDI file does not specify a starting **tempo**, the default value of 120 beats per minute shall be used.

MIDI time-stamps may only appear in the “MIDI Time Code” syntax, not the “SMPTE” syntax, as described in [MIDI]. The SMPTE Offset meta-event and the SMPTE format delta-time object are not supported in Structured Audio Object 3 and Object 4 bitstreams.

As events are converted from MIDI to SAOL semantics, each event shall be registered with the scheduler according to its event time and semantics.

NOTE - MIDI tempo events are not used to calculate the event time of MIDI events in a MIDI file. Rather, they are scheduled as regular events and dispatched according to the tempo semantics in 5.7.3.3.6 list item 7.

5.14.3.3.4 Converting MIDI channels to scheduler channels

Mapping the channels of events in MIDI files to **midi_event channel** numbers is accomplished as follows. Successive track chunks within a **midi_file** are assigned track numbers in ascending monotonic sequence, with an initial track number of 0. The **channel** value for a particular **midi_file** event is given by:

channel = **midi_file** event channel number + (track number * 16).

If there are multiple **midi_file** chunks within the bitstream header, then subsequent **midi_file** elements have tracks numbered sequentially from the end of the first chunk. That is, if the i^{th} **midi_file** element has k_i tracks, $0 < i < n$, then the tracks from the first **midi_file** are numbered $0..k_0\text{-}1$, those from the second are numbered $k_0..k_1\text{-}1$, and so forth.

EXAMPLE

A **midi_file** containing ten track chunks would be mapped onto a set of 160 **midi_event channel** values. Events in track chunk 0 would map to **channel** values in the range 0…15. Events in track chunk 1 would map to **channel** values in the range 16…31. Events in track chunk 9 would map to **channel** values in the range 144…159.

If some channel numbers within a given track chunk are unused, the corresponding **channel** values are also unused.

5.14.3.4 Default controller values

The following table gives the default values for certain continuous controllers. If a particular controller is not listed here, then its default value shall be zero.

There is no normative significance to these "function names" excepting controller 64; however, content authors who wish to use General MIDI score files with SAOL orchestras are advised to consult **[MIDI]** for the normative meaning of the controllers and controller values within General MIDI bitstreams and MIDIfiles.

Table 5.4 — Default MIDI Controller Values

Controller	Function	Default
1	Mod Wheel	0
5	Portamento Speed	0
7	Volume	100
10	Pan	64
11	Expression	127
64	Sustain Pedal	0
65	Portamento On/Off	0
66	Sostenuto	0
67	Soft Pedal	0
84	Portamento Control	0
Pitch Bend	Pitch Bend	8192

5.15 Input sounds and relationship with AudioBIFS

5.15.1 Introduction

This subclause describes the use of SAOL orchestras as the effects-processing functionality in the AudioBIFS (Binary Format for Scene Description) system, described in ISO/IEC 14496-1 subclause 9.2.2.13.3. In ISO/IEC 14496, SAOL is used not only as a sound-synthesis description method, but also as a description method for sound-effects and other post-production algorithms. The BIFS **AudioFX** node (ISO/IEC 14496-1, subclause 9.4.2.7) allows the inclusion of signal-processing algorithms described in SAOL that are applied to the outputs of the sound nodes subsidiary to that node in the scene graph. This functionality fits well into the bus-send methodology in Structured Audio, but requires some additional normative text to exactly describe the process.

5.15.2 Input sources and phaseGroup

Each node in a BIFS scene graph that contains SAOL code is either an **AudioSource** node or an **AudioFX** node. If the former, there are no input sources to the SAOL orchestra, and so the default orchestra global **inchannels** value is 0 (see subclause 5.8.5.2.3). In this case, the special bus **input_bus** may not be sent to an instrument or otherwise used in the orchestra.

If the latter, the child nodes of the **AudioFX** node provide several channels of input sound to the orchestra. These channels of input sound, calculated as described in ISO/IEC 14496-1 subclause 9.4.2.7, are placed on the special bus **input_bus**. From this bus, they may be sent to any instrument(s) desired and the audio data thereby provided shall be treated normally. The number of orchestra input channels---the default value of orchestra global **inchannels**---is the sum of the numbers of channels of sound provided by each of the children.

In any instrument that receives a **send** from the special bus **input_bus**, the value of the **inGroup** standard name (see subclause 5.8.6.8.15) shall be constructed using the **phaseGroup** values of the child nodes in the scene graph, as follows. The **inGroup**[] values, when non-zero, shall have the property that **inGroup**[**i**] **= inGroup**[**j**] when **i** ≠ **j** exactly when **input** channel **i** is output channel **n** of child **c1**, **input** channel **j** is output channel **m** of child **c2**, **c1 = c2**, and **phaseGroup**[**n**] = **phaseGroup**[**m**] within **c1**. (That is, when the two channels come from the same child and are phase-grouped in that child's output).

This rule applies in addition to the usual **inGroup** rules as given in subclause 5.8.6.8.15.

NOTE

The **phaseGroup** values are made available to the SA decoder, but they are not visible inside the scope of the several orchestra elements, and therefore **phaseGroup** does not constitute a standard name as **inGroup**. **phaseGroup** is only used outside the orchestra by the SA decoder to construct the **inGroup** values.

EXAMPLE

Assume that the two child nodes of an **AudioFX** node produce two and three channels of output respectively, and their **phaseGroup** fields are [1,1] and [1,0,1] respectively. That is, in the first child, the two channels form a stereo pair; and in the second, the first and third channels form a stereo pair that has no phase relationships with the second channel.

For the following global orchestra definitions:

```
send(input_bus ; ; a);
route(a, bus2);
send(bus2,input_bus,bus2 ; ; b);
```

Assume that instrument **a** produces two channels of output. Then, a legal value for the **inGroup** name within **a** is [1,1,2,0,2], and a legal value for the **inGroup** name within **b** is [1,1,2,2,3,0,3,4,4]. The value for the **inGroup** name within **a** shall not be [1,1,1,0,1], and the value for the **inGroup** name within **b** shall not be [1,1,2,2,2,2,2,3,3] (among other illegal possibilities).

5.15.3 The AudioFX node

5.15.3.1 Introduction

The **AudioFX** node in AudioBIFS is described in the MPEG-4 Systems document, ISO/IEC 14496-1, subclause 9.4.2.7. It is used therein to download audio effects-processing algorithms within the AudioBIFS toolset. SAOL is the language for description of audio effects-processing algorithms in AudioBIFS. This subclause describes the execution of a Structured Audio orchestra for the purpose of processing sounds in the AudioBIFS scene.

The aspects described in this subclause are normative, but the behaviour is not “decoding” behaviour. Rather, this subclause expands the normative processing semantics of the **AudioFX** node description in ISO/IEC 14496-1.

5.15.3.2 AudioFX orchestra parameters

When a SAOL orchestra is instantiated due to an **AudioFX** BIFS node, only an orchestra file (the **orch** field in the node) and, optionally, a SASL score file (the **score** field) are provided. These files correspond to tokenised sequences of orchestra and score data forming legal **orchestra** and **score_file** bitstream elements as described in subclause 5.5.2. Further, a score may not contain new instrument events, but only control parameters for the **send** instruments defined in the orchestra. The set of allowable sampling rates is restricted, see subclause 5.8.5.2.1.

5.15.3.3 AudioFX orchestra instantiation

To instantiate the orchestra for the AudioFX node requires the following steps:

1. Decoding of the **orch** and **score** (if any) elements in the node
2. Parsing and syntax-checking of these elements
3. Instantiation of **send** instruments in the orchestra (as described in subclause 5.7.2).

Each of these **send** instances shall be maintained until it is turned off by the **turnoff** statement, or the node containing the orchestra is deleted from the scene graph. If the **turnoff** statement is used in one of these instruments, it shall be taken as producing zero values for all future time.

5.15.3.4 AudioFX orchestra execution

The run-time synthesis process proceeds according to the rules cited in subclause 5.7.3.3 for a standard SA decoding process, with the following exceptions and additions:

As no access units will be received by an **AudioFX** process, no communication with the systems layer need be maintained for this purpose. The only events used are those that are in the **score** field of the node itself. At each time step, the **AudioFX** orchestra shall request from the systems layer the input audio buffers that correspond to the child nodes. These audio buffers shall be placed on the special bus **input_bus** and then sent to whatever instruments are specified in the global orchestra header.

Also, at each control-rate step, the **params[]** fields of the **AudioFX** node shall be copied into the global **params[]** array of the orchestra. These fields are exposed in the scene graph so that interactive aspects of other parts of the scene graph may be used to control the orchestra. At the end of each control cycle, the **params[]** array values shall be copied back into the corresponding fields of the **AudioFX** node and then routed to other nodes as specified within the scene graph. (It is not possible to give a more semantically meaningful field name than **params** since the purpose of the field may vary greatly from application to application, depending on the needs of the content).

At every point in time, the output of the orchestra becomes the output of the **AudioFX** node.

5.15.3.5 Speed change functionality in the AudioFX node

Speed change functionality for sounds provided from the input sources is supported in the AudioFX node. The SAOL core opcode **fx_speedc** is provided for this purpose; see subclause 5.9.14.4.

5.15.4 Interactive 3-D spatial audio scenes

When an **AudioSource** or **AudioFX** node is the child of a **Sound** node, the spatial location, direction, and propagation pattern of the sound subtree represented at the position of the **Sound** node, and the spatial location and direction of the listener, are provided to the SAOL code in the node. In this way, subtle spatial effects such as source directivity modelling may be written in SAOL.

The standard names **position**, **direction**, **listenerPosition, listenerDirection, minFront, maxFront, minBack,** and **maxBack** (see subclauses 5.8.6.8.18-5.8.6.8.25) are used for this purpose.

It is not recommended that content providing 3-D spatial audio in the context of audio-visual virtual reality applications in BIFS use the **spatialize** statement within SAOL to provide this functionality. In most terminals, the scene-composition 3-D audio functionality will be able to use more information about the interaction process to provide the best-quality audio rendering. In particular, spatial positioning and source directivity are implemented at the end terminal with a sophistication suitable for the terminal itself (see ISO/IEC 14496-1, **Sound** node specification, subclause 9.4.2.82). Content authors can use SAOL and the **AudioFX** node to create enhanced spatial effects that include reverberation, environmental attributes and complex attenuation functions, and then let the terminal-level spatial audio presentation be used to interface with the available rendering method for the terminal. The **spatialize** statement in SAOL is provided for the creation of non-interactive spatial audio effects in musical compositions, so that composers may tightly integrate the spatial presentation with other aspects of the musical material.

Annex 5.A
(normative)

Coding tables

5.A.1 Introduction

This Annex contains the bitstream token table as referenced in subclause 5.5 and subclause 5.12. Certain tokens are indicated as **(reserved),** which means they are not currently used in the bitstream, but may be used in future versions of the standard. Tokens 0xF5 through 0xFF may be used by implementers for implementation-dependent purposes.

5.A.2 Bitstream token table

Token	Text
0x00	**(reserved)**
0x01	aopcode
0x02	asig
0x03	else
0x04	exports
0x05	extend
0x06	global
0x07	if
0x08	imports
0x09	inchannels
0x0A	instr
0x0B	iopcode
0x0C	ivar
0x0D	kopcode
0x0E	krate
0x0F	ksig
0x10	map
0x11	oparray
0x12	opcode
0x13	outbus
0x14	outchannels
0x15	output
0x16	return
0x17	route
0x18	send
0x19	sequence
0x1A	sasbf
0x1B	spatialize
0x1C	srate
0x1D	table
0x1E	tablemap
0x1F	template
0x20	turnoff
0x21	while
0x22	with
0x23	xsig
0x24	interp
0x25	preset
0x26-0x2F	**(reserved)**
0x30	k_rate
0x31	s_rate
0x32	inchan
0x33	outchan
0x34	time
0x35	dur
0x36	MIDIctrl
0x37	MIDItouch
0x38	MIDIbend
0x39	input
0x3A	inGroup
0x3B	released
0x3C	cpuload
0x3D	position
0x3E	direction
0x3F	listenerPosition
0x40	listenerDirection
0x41	minFront
0x42	minBack
0x43	maxFront
0x44	maxBack
0x45	params
0x46	itime
0x47	**(reserved)**
0x48	channel
0x49	input_bus
0x4A	output_bus
0x4B	startup
0x4C-0x4F	**(reserved)**
0x50	&&
0x51	||
0x52	>=
0x53	<=
0x54	!=
0x55	==
0x56	-
0x57	*
0x58	/
0x59	+
0x5A	>
0x5B	<
0x5C	?
0x5D	:
0x5E	(
0x5F	)
0x60	{
0x61	}
0x62	[
0x63	]
0x64	;
0x65	,
0x66	=
0x67	!
0x68-0x6E	**(reserved)**
0x6F	sample
0x70	data
0x71	random
0x72	step
0x73	lineseg
0x74	expseg
0x75	cubicseg
0x76	polynomial
0x77	spline
0x78	window
0x79	harm
0x7A	harm_phase
0x7B	periodic
0x7C	buzz
0x7D	concat
0x7E	empty
0x7F	**(reserved)**
0x80	int
0x81	frac
0x82	dbamp

0x83	ampdb
0x84	abs
0x85	exp
0x86	log
0x87	sqrt
0x88	sin
0x89	cos
0x8A	atan
0x8B	pow
0x8C	log10
0x8D	asin
0x8E	acos
0x8F	floor
0x90	ceil
0x91	min
0x92	max
0x93	pchoct
0x94	octpch
0x95	cpspch
0x96	pchcps
0x97	cpsoct
0x98	octcps
0x99	pchmidi
0x9A	midipch
0x9B	octmidi
0x9C	midioct
0x9D	cpsmidi
0x9E	midicps
0x9F	sgn
0xA0	ftlen
0xA1	ftloop
0xA2	ftloopend
0xA3	ftsetloop
0xA4	ftsetend
0xA5	ftbasecps
0xA6	ftsetbase
0xA7	tableread
0xA8	tablewrite
0xA9	oscil
0xAA	loscil
0xAB	doscil
0xAC	koscil
0xAD	kline
0xAE	aline
0xAF	sblock
0xB0	kexpon
0xB1	aexpon
0xB2	kphasor
0xB3	aphasor
0xB4	pluck
0xB5	buzz
0xB6	grain
0xB7	irand
0xB8	krand
0xB9	arand
0xBA	ilinrand
0xBB	klinrand
0xBC	alinrand
0xBD	iexprand
0xBE	kexprand
0xBF	aexprand
0xC0	kpoissonrand
0xC1	apoissonrand
0xC2	igaussrand
0xC3	kgaussrand
0xC4	agaussrand
0xC5	port
0xC6	hipass
0xC7	lopass
0xC8	bandpass
0xC9	bandstop
0xCA	fir
0xCB	iir
0xCC	firt
0xCD	iirt
0xCE	biquad
0xCF	fft
0xD0	ifft
0xD1	rms
0xD2	gain
0xD3	balance
0xD4	decimate
0xD5	upsamp
0xD6	downsamp
0xD7	samphold
0xD8	delay
0xD9	delay1
0xDA	fracdelay
0xDB	comb
0xDC	allpass
0xDD	chorus
0xDE	flange
0xDF	reverb
0xE0	compressor
0xE1	gettune
0xE2	settune
0xE3	ftsr
0xE4	ftsetsr
0xE5	gettempo
0xE6	settempo
0xE7	fx_speedc
0xE8	speedt
0xE9-0xEF	**(reserved)**
0xF0	<symbol>
0xF1	<number>
0xF2	<integer>
0xF3	<string>
0xF4	<byte>
0xF5-0xFF	**(free)**
0xFF	<EOO>

Annex 5.B
(informative)

Encoding

5.B.1 Introduction

This Annex, provided for informative purposes only, provides guidelines for building a typical Structured Audio encoder. Unlike for the natural audio coders described in Subparts 2, 3 and 4, at the time of the completion of ISO/IEC 14496, there is no known technique for generally and automatically encoding Structured Audio bitstreams from acoustic data. The computational methods that would be required to accomplish this—known variously as "computational auditory scene analysis", "automatic polyphonic transcription", and "musical acoustic source separation"—are still in a research stage and not likely to be generally available for several years.

Thus, the creation of bitstreams complying to this subclause is a process that requires human intervention and assistance. This Annex describes the functions of possible tools for creating such bitstreams; the techniques described here are for informative purposes only, and are not required for compliance to ISO/IEC 14496-3.

5.B.2 Basic encoding

5.B.2.1 Introduction

This subclause describes the operation of a *basic encoder* of the sort provided with IS0/IEC 14496-5. A basic Structured Audio encoder takes as input the component units of a bitstream conforming to the description in subclause 5.5 (such as orchestra files and sound samples), and converts them into in a legal bitstream representation. It is outside the scope of this Annex to discuss the origin of the component units themselves; perhaps they have been created by hand or with the use of other general-purpose computer tools.

For the purposes of this discussion, it is assumed that the component units are in the following formats: SAOL and SASL programs are in their respective textual formats as described in subclause 5.8 and subclause 5.11 respectively; sound samples are individually stored in computer sound-file format such as AIFF or WAVE; MIDI data is stored as a Standard MIDI File; and SASBF banks are stored as binary data files.

The steps required in bitstream creation are as follows: tokenisation of the SAOL and SASL programs, disassembly of the sound samples, assembly of the decoder configuration information, and (optionally) reorganisation of the score and MIDI events into streaming data. Each of these steps is described in the following subclauses.

5.B.2.2 Tokenisation of SAOL data

The tokenisation of SAOL data is conducted as described in subclause 5.12.2. This process converts the SAOL program given in the textual format into a binary block of data. During this process, a *symbol table* may be constructed if desired by enumerating the names of the instruments, user-defined opcodes, wavetables, and signal variables in the orchestra, and associating each with a numeric value. This table may be incorporated in the decoder configuration header of the bitstream as described in subclause 5.5.2; it has no normative significance in decoding, but allows human-readable SAOL and SASL programs in the textual format to be recovered from the bitstream.

5.B.2.3 Tokenisation of SASL data

The tokenisation of SASL data is conducted as described in subclause 5.12.3. This process converts the SASL program given in the textual format into a binary block of data. Any symbols used in the SASL score may be

incorporated into the symbol table if one was constructed in the process described in subclause 5.B.2.2; however, at most one symbol table may be used in the orchestra.

5.B.2.4 Disassembly of sound samples

The sound samples stored as computer sound files are disassembled into blocks of sample values. It is not permissible to include sound samples with formatted data (such as an AIFF or WAVE file) directly in the Structured Audio bitstream. The length (in samples), sampling rate (in Hz) if available, base frequency (in Hz) if needed, and loop start and end points (in sample number) if needed are accounted from the formatted information in the computer sound file. The sound samples are converted from whatever format they were stored in the computer sound file into either 16-bit signed integer values (that is, the values are scaled into the range [-32768, 32767]) or into 32-bit signed floating point values. Either format may be used for a sample in the Structured Audio bitstream; the first is more efficient and the second is more precise.

NOTE - If very long sound samples are to be used, and one or more natural sound decoders (that is, the functionality described in clauses 2, 3, and 4) are present in the terminal, the natural sound decoders may be used to compress sound samples as described in subclause 5.10.2 of this document and subclause 9.4.2.4 of ISO/IEC 14496-1. In this case, the sound sample(s) is (are) not contained in the Structured Audio bitstream, but in one or more natural audio bitstreams that are associated with the Structured Audio bitstream through use of the **AudioBuffer** AudioBIFS node as described in subclause 9.4.2.4 of ISO/IEC 14496-1. This process can result in more efficient transmission for Structured Audio bitstreams containing long samples.

5.B.2.5 Assembly of decoder configuration information

The decoder configuration header is constructed according to the format given in subclause 5.5 from a tokenised SAOL orchestra, and possibly one or more tokenised SASL scores, sound samples, MIDI files, and SASBF banks. The blocks may be in any desired order, and are indexed by the **more_data** and **chunk_type** bit fields. The **high_priority** bit of each score event in the header may be set for each score line in the score that contained the ***** tag, or for any set of important events desired.

5.B.2.6 Assembly of streaming bitstream

In the Structured Audio bitstream format, streaming data in the form of access units is not strictly required; all of the information required for decoding may be present in the decoder configuration header. Including streaming data in the form of access units may make it easier for general-purpose MPEG-4 tools to reorganise the bitstream data for the purposes of editing or resynchronisation, or to execute random-access control of the bitstream (see Annex 5.C).

Sound samples, score events, and MIDI commands may all be included in the streaming-data part of the Structured Audio bitstream. A sound sample is included simply by packaging the sound data, after it has been disassembled from the computer sound file format, into an access unit as specified in subclause 5.5.2. A score event may be included with or without a timestamp as discussed in subclause 5.5.2. If it is included with a timestamp, it is subject to internal orchestra tempo control as discussed in subclause 5.7.3.3.6, but is difficult to reschedule at the Access Unit level; if it is included without timestamps, it is easier to reschedule at the Access Unit level, and is not subject to score-based control of the tempo. If there is no explicit timestamp, the event timing is controlled by the synchronisation information in the Access Unit, see subclause 7.2.3 of ISO/IEC 14496-1. In this case, the **use_if_late** tag may be used to indicate whether the event shall be used if it arrives late; see subclause 5.7.3.3.8.

For each score event, the **high_priority** bit may be set if the corresponding line in the score contained the ***** tag, or for any set of important events desired.

MIDI commands are first converted from the Standard MIDI File format to MIDI data representations. To accomplish this, the absolute time of each event in the Standard MIDIFile is computed according to the syntax and semantics in [MIDI]. Then, the events are not included with delta-times, but are placed directly in the Access Unit, so that the synchronisation information in the Access Unit controls the event timing for the MIDI events. The MIDI data in each Access Unit in the bitstream is the same as that that is conveyed in the MIDI protocol in real-time MIDI-based performance; that is, it consists of un-timestamped note on, note off, and controller information.

MIDI events are wrapped in the length indicator as indicated in subclause 5.5; the format of streaming MIDI data in MPEG-4 is not bit-for-bit identical (and thus, not as compact) as MIDI data in the strict MIDI protocol as specified in **[MIDI]**, subclause 5.2.

NOTE - The streaming data constructed by the process is noted to be “bursty” and highly variable-rate. That is, when a large access unit is conveyed, for example, a sound sample, the effective bitrate is suddenly much higher than when only small access units such as note events are conveyed. The Access-Unit repackaging techniques described in clauses 10 and 11 of ISO/IEC 14496-1 may be used to smooth out the bitrate of the Structured Audio bitstream.

Annex 5.C
(informative)

lex/yacc grammars for SAOL

5.C.1 Introduction

This Annex provides grammars using the widely-available tools ‘lex’ and ‘yacc’ that conform to the SAOL specification in this document. They are provided for informative purposes only; implementers are free to use whichever tools they desire, or no tools, in building an implementation.

The reference software for Structured Audio in ISO/IEC 14496-5 builds the lexer and parser for SAOL out of these grammars by augmenting them with more processing and data-structures.

5.C.2 Lexical grammar for SAOL in lex

```
STRCONST    \"(\\.|[^\\"])*\"
IDENT       [a-zA-Z_][a-zA-Z0-9_]*
INTGR       [0-9]+
NUMBER      [0-9]+(\.[0-9]*)?(e[-+]?[0-9]+)?|-?\.[0-9]*(e-+?[0-9]+)?

%{
void comment(void);
%}

%%

"//"          { comment(); }
"aopcode"     { return(AOPCODE) ; }
"asig"        { return(ASIG) ; }
"else"        { return(ELSE) ; }
"exports"     { return(EXPORTS) ; }
"extend"      { return(EXTEND) ; }
"global"      { return(GLOBAL) ; }
"if"          { return(IF) ; }
"imports"     { return(IMPORTS); }
"inchannels"  { return(INCHANNELS) ; }
"instr"       { return(INSTR); }
"interp"      { return(INTERP); }
"iopcode"     { return(IOPCODE); }
"ivar"        { return(IVAR) ; }
"kopcode"     { return(KOPCODE); }
"krate"       { return(KRATE) ; }
"ksig"        { return(KSIG) ; }
"map"         { return(MAP) ; }
"oparray"     { return(OPARRAY) ; }
"opcode"      { return(OPCODE) ; }
"outbus"      { return(OUTBUS) ; }
"outchannels" { return(OUTCHANNELS) ; }
"output"      { return(OUTPUT) ; }
"preset"      { return(PRESET) ; }
"return"      { return(RETURN) ; }
"route"       { return(ROUTE) ; }
"send"        { return(SEND) ; }
"sequence"    { return(SEQUENCE) ; }
"sasbf"       { return(SASBF) ; }
```

```
"spatialize"  { return(SPATIALIZE) ; }
"srate"       { return(SRATE); }
"table"       { return(TABLE); }
"tablemap"    { return(TABLEMAP); }
"template"    { return(TEMPLATE); }
"turnoff"     { return(TURNOFF); }
"while"       { return(WHILE); }
"with"        { return(WITH); }
"xsig"        { return(XSIG) ; }
"&&"          { return(AND); }
"||"          { return(OR); }
">="          { return(GEQ); }
"<="          { return(LEQ); }
"!="          { return(NEQ); }
"=="          { return(EQEQ); }
"-"           { return(MINUS); }
"*"           { return(STAR); }
"/"           { return(SLASH); }
"+"           { return(PLUS); }
">"           { return(GT); }
"<"           { return(LT); }
"?"           { return(Q); }
":"           { return(COL); }
"("           { return(LP); }
")"           { return(RP); }
"{"           { return(LC); }
"}"           { return(RC); }
"["           { return(LB); }
"]"           { return(RB); }
";"           { return(SEM); }
","           { return(COM); }
"="           { return(EQ); }
"!"           { return(NOT); }

{STRCONST}    { yytext[strlen(yytext)-1] = 0; /* strip quotes */
                yylval = strdup(&yytext[1]);
                return(STRCONST); }
{IDENT}       { yylval = strdup(yytext);
                return(IDENT) ; }
{INTGR}       { yylval = strdup(yytext);
                return(INTGR) ; }
{NUMBER}      { yylval = strdup(yytext);
                return(NUMBER) ; }
[ \t\n\r]     { /* whitespace */ }

.             { printf("Line %d: Unknown character: '%s'\n",
                        yyline,yytext); } /* parse error */

%%

void comment() {
  char c;

  while ((c = input()) != '\n'); /* skip */
  yyline++;
  thisline[0] = 0;
  yycol = 0;
  }
```

5.C.3 Syntactic grammar for SAOL in yacc

```
/*

   This is a grammar for SAOL, written in 'yacc'.

   It has one shift/reduce conflict that arises when
   looking at table definitions.  Within the grammar,
   'table t1;' is allowed as a definition even though
   it is  a prohibited construction (it is flagged in the syntax-
   check).  So there's an ambiguity between

     table t1(...)
     table t1;

  and you don't know with one lookahead where you are when you
  get 'table' if 't1' is the next token.

  This grammar is somewhat less strict than it could be about
  creating parse errors for all syntax errors.  As it is used
  in ISO/IEC 14496-5, many illegal constructions are allowed here,
  but then prohibited in the syntax check.

  The code in ISO/IEC 14496-5 for Structured Audio uses this grammar
  as a basis, and augments it with error productions and construction
  of a parse tree.

  Note that this grammar does not fully support the template construct
  as defined in subclause 5.8.8.2. This subclause permits expressions in
  the preset and map lists, whereas this grammar only supports terminals
  in these lists. To support the full template syntax, a lexical analyzer
  that detects expressions in preset and map lists and takes special action
  is necessary.

*/

%token IDENT INTGR NUMBER STRCONST AOPCODE ELSE EXPORTS EXTEND GLOBAL
%token IF IMPORTS INCHANNELS INTERP
%token INSTR IOPCODE IVAR TABLE KOPCODE KRATE KSIG ASIG MAP
%token OPARRAY OPCODE OUTBUS OUTCHANNELS OUTPUT ROUTE SEND SEQUENCE
%token SRATE TEMPLATE TURNOFF WHILE WITH XSIG AND OR GEQ LEQ
%token NEQ EQEQ MINUS STAR SPATIALIZE SASBF TABLEMAP
%token SLASH PLUS GT LT Q COL LP RP LC RC LB RB SEM COM EQ RETURN NOT
%token ARRAYREF OPCALL IMPEXP VARDECL NOTAG SPECIALOP PRESET

%

%start orcfile
%right Q
%left OR
%left AND
%left EQEQ NEQ
%left LT GT LEQ NEQ
%left PLUS MINUS
%left STAR SLASH
%right UNOT UMINUS
%token HIGHEST

%%

orcfile       : proclist
              ;

proclist      : proclist instrdecl
              | proclist opcodedecl
              | proclist globaldecl
```

```
              | proclist templatedecl
              | /* null */
              | error
              ;

instrdecl     : INSTR IDENT LP identlist RP miditag LC vardecls block RC
              ;

miditag       : PRESET int_list
              | /* null */
              ;

int_list      : int_list INTGR
              | INTGR
              ;

opcodedecl    : optype IDENT LP paramlist RP LC opvardecls block RC
              ;

globaldecl    : GLOBAL LC globalblock RC
              ;

templatedecl  : TEMPLATE LT identlist GT  /* with preset */
                 PRESET mapblock
                 LP identlist RP
                 MAP LC identlist RC
                 WITH LC mapblock RC LC
                 vardecls block RC
              | TEMPLATE LT identlist GT  /* no preset */
                 LP identlist RP
                 MAP LC identlist RC
                 WITH LC mapblock RC LC
                 vardecls block RC
              ;

mapblock      : mapblock COM LT terminal_list GT
              | LT terminal_list GT
              |
              ;

terminal_list : terminal_list COM terminal
              | terminal
              ;

terminal      : IDENT
              | const
              | STRCONST
              ;

globalblock   : globalblock globaldef
              | /* null */
              ;

globaldef     : rtparam
              | vardecl
              | routedef
              | senddef
              | seqdef
              ;

rtparam       : SRATE INTGR SEM
              | KRATE INTGR SEM
              | INCHANNELS INTGR SEM
              | OUTCHANNELS INTGR SEM
```

```
              | INTERP INTGR SEM
              ;

routedef      : ROUTE LP IDENT COM identlist RP SEM
              ;

senddef       : SEND LP IDENT SEM exprlist SEM namelist RP SEM
              ;

seqdef        : SEQUENCE LP identlist RP SEM
              ;

block         : block statement
              | /* null */
              ;

statement     : lvalue EQ expr SEM
              | expr SEM
              | IF LP expr RP LC block RC
              | IF LP expr RP LC block RC ELSE LC block RC
              | WHILE LP expr RP LC block RC
              | INSTR IDENT LP exprlist RP SEM
              | OUTPUT LP exprlist RP SEM
              | SPATIALIZE LP exprlist RP SEM
              | OUTBUS LP IDENT COM exprlist RP SEM
              | EXTEND LP expr RP SEM
              | TURNOFF SEM
              | RETURN LP exprlist RP SEM
              ;

lvalue        : IDENT
              | IDENT LB expr RB
              ;

identlist     : identlist COM IDENT
              | IDENT
              | /* null */
              ;

paramlist     : paramlist COM paramdecl
              | paramdecl
              |   /* null */
              ;

vardecls      : vardecls vardecl
              | /* null */
              ;

vardecl       : taglist stype namelist SEM
              | stype namelist SEM
              | tabledecl SEM
              | TABLEMAP IDENT LP identlist RP SEM
              ;

opvardecls    : opvardecls opvardecl
              | /* null */
              ;

opvardecl     : taglist otype namelist SEM
              | otype namelist SEM
              | tabledecl SEM
              | TABLEMAP IDENT LP identlist RP SEM
              ;
```

```
paramdecl     : otype name
              ;

namelist      : namelist COM name
              | name
              ;

name          : IDENT
              | IDENT LB INTGR RB
              | IDENT LB INCHANNELS RB
              | IDENT LB OUTCHANNELS RB
              ;

stype         : IVAR
              | KSIG
              | ASIG
              | TABLE
              | OPARRAY
              ;

otype         : XSIG
              | stype
              ;

tabledecl     : TABLE IDENT LP IDENT COM exprstrlist RP
              ;

taglist       : IMPORTS
              | EXPORTS
              | IMPORTS EXPORTS
              | EXPORTS IMPORTS
              ;

optype        : AOPCODE
              | KOPCODE
              | IOPCODE
              | OPCODE
              ;

expr          : IDENT
              | const
              | IDENT LB expr RB
              | SASBF LP exprlist RP
              | IDENT LP exprlist RP
              | IDENT LB expr RB LP exprlist RP
              | expr Q expr COL expr %prec Q
              | expr LEQ expr
              | expr GEQ expr
              | expr NEQ expr
              | expr EQEQ expr
              | expr GT expr
              | expr LT expr
              | expr AND expr
              | expr OR expr
              | expr PLUS expr
              | expr MINUS expr
              | expr STAR expr
              | expr SLASH expr
              | NOT expr     %prec UNOT
              | MINUS expr
              | LP expr RP
              ;

exprlist      : exprlist COM expr
```

```
              | expr
              | /* null */
              ;

/* this is used for table declarations; string constants provide
   above-the-standard file handling for “sample” */

exprstrlist   : exprstrlist COM expr
              | exprstrlist COM STRCONST
              | STRCONST
              | expr
              ;

const         : INTGR
              | NUMBER
              ;
```

Annex 5.D
(informative)

PICOLA Speed change algorithm

5.D.1 Tool description

PICOLA (Pointer Interval Controlled OverLap Add) speed control tool supports speed change functionality for mono-channel signals. The speed control is achieved by replacing a part of the input signal with an overlap-added waveform or by inserting the overlap-added waveform into the input signal.

5.D.2 Speed control process

The block diagram of the speed controller is shown in Figure 5.D.1. The input signal InputSignal, which is an output from the audio source with a given frame length NumInputSample, is stored in the buffer memory. Adjacent waveforms with the same length are extracted in pairs from the memory buffer and the pair with the minimum distortion between the two waveforms is selected. The selected waveforms are overlap-added. The speed control is achieved by replacing a part of the input signal with the overlap-added waveform or by inserting it into the input signal. The speed controller outputs the speed changed signal with a certain fixed length frame calculated as NumInputSample /SpeedControlFactor. Details of the processing are described below.

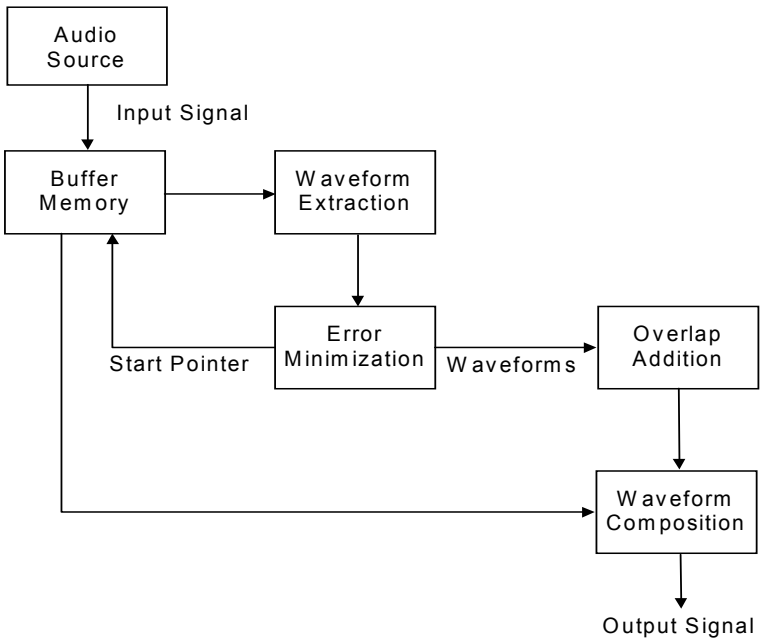


Figure 5.D.1 — Block Diagram of the Speed Controller

5.D.3 Time scale compression (High speed replay)

The compression principle is shown in Figure 5.D.2. P0 is the pointer that indicates the starting sample of the current processing frame in the memory buffer. The processing frame has a length of LF samples and comprises adjacent waveforms each length of LW samples. The average distortion per sample between the first half of the processing frame (waveform A) and the second half (waveform B) is calculated as shown below.

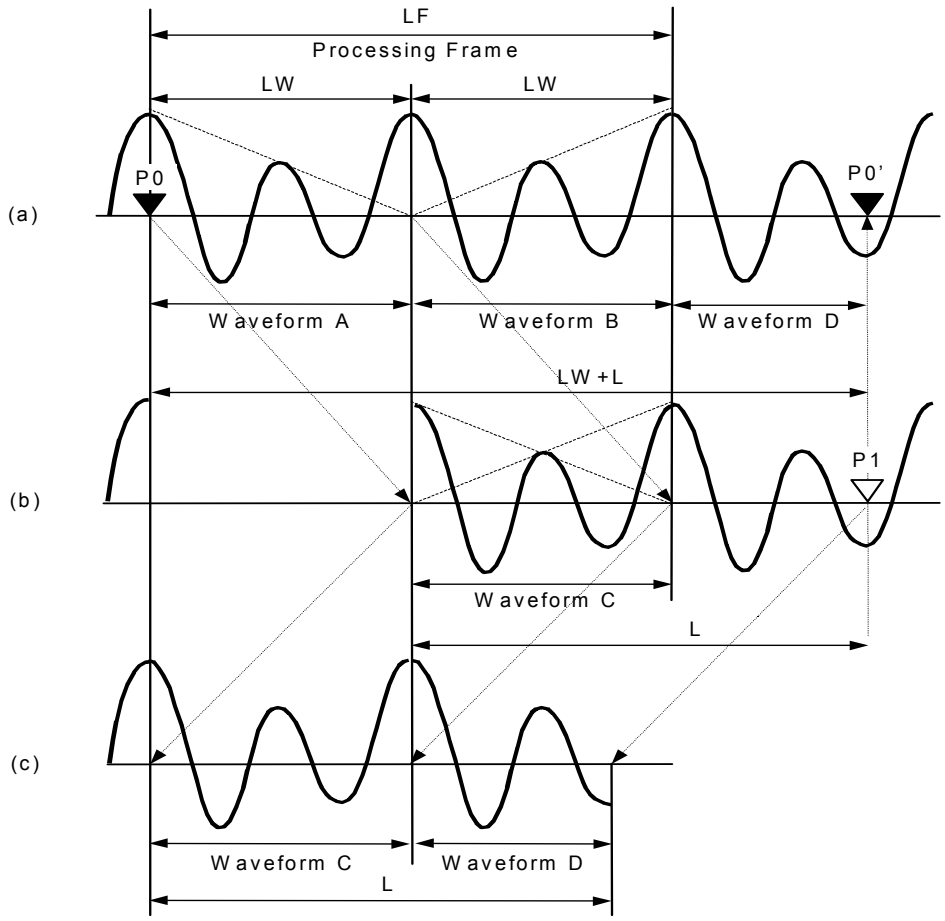
$$D(LW) = \frac{1}{LW}\sum_{n=0}^{LW-1}\{x(n) - y(n)\}^2 \qquad (PMIN \le LW \le PMAX)$$

where D(LW) is the average distortion between the two waveforms when the waveform length is LW, $x(n)$ is the waveform A, $y(n)$ is the waveform B, PMIN is the minimum length of the candidate waveform and PMAX is the maximum length of the candidate waveform. Typically PMIN=32 and PMAX=160 for 8kHz sampling rate, PMIN=80 and PMAX=320 for 16kHz sampling.

The length LW that minimizes the distortion D(LW) is selected, and corresponding waveforms A and B are determined. If the cross-correlation between the selected waveforms A and B is negative, the length LW is set to PMIN-1. After the waveform length LW is determined, the waveform A is windowed by a half triangular window with a descending slope, and the waveform B is windowed by a half triangular window with an ascending slope. The overlap-added waveform C is obtained by linearly adding the windowed waveform A and waveform B. Then, the pointer P0 moves to the point P1. The distance L from the beginning of waveform C to the pointer P1 is given by;

$$L = LW \bullet \frac{1}{SpeedControlFactor - 1} \qquad (SpeedControlFactor > 1)$$

L samples from the beginning of waveform C are output as the compressed signal. If L is greater than LW, the original waveform D that follows the waveform B is also output. Therefore the length of the signal is shortened from LW+L samples to L samples. The updated pointer P1 indicates the starting sample P0' of the next processing frame.



(a) Original signal; (b) Overlap-added waveform; (c) Compressed signal

Figure 5.D.2 — Principle of Time Scale Compression

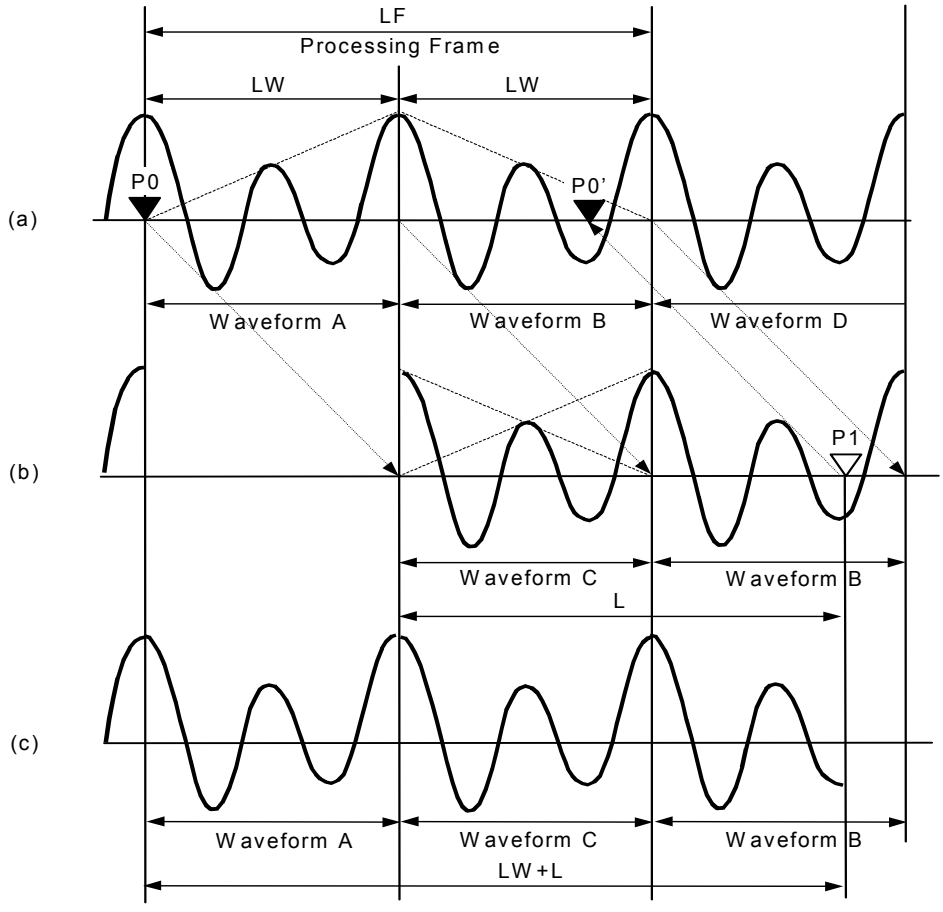
5.D.4 Time scale expansion (Low speed replay)

The expansion principle is shown in (a) Original signal; (b) Overlap-added waveform; (c) Expanded signal

Figure 5.D.3. P0 is the pointer that indicates the starting sample of the current processing frame in the memory buffer. The processing frame has a length of LF samples and includes adjacent waveforms each length of LW samples. After the waveform length LW is determined using the same method as described in the time scale compression, the first half of the processing frame (waveform A) is outputted without any modification. Next, the first half (waveform A) is windowed by a half triangular window with an ascending slope, and the second half (waveform B) is windowed by a half triangular window with a descending slope. The overlap-added waveform C is obtained by linearly adding the windowed waveform A and waveform B. Then, the pointer P0 moves to the point P1. The distance L from the beginning of waveform C to the pointer P1 is given by;

$$L = LW \bullet \frac{SpeedControlFactor}{1 - SpeedControlFactor} \qquad (0.5 \le SpeedControlFactor < 1)$$

L samples from the beginning of waveform C are output as the expanded signal. If L is greater than LW, the original waveform B is repeated as the output. The length of the signal is therefore expanded from L samples to LW+L samples. The updated pointer P1 indicates the starting sample P0' of the next processing frame.



(a) Original signal; (b) Overlap-added waveform; (c) Expanded signal

Figure 5.D.3 — Principle of Time Scale Expansion

Annex 5.E
(informative)

Random access to Structured audio bitstreams

5.E.1 Introduction

This Annex describes practices for constructing Structured Audio bitstreams that allow easy random access to sound content. As discussed in subclause 5.E.2, not every sound described with a bitstream conforming to the Structured Audio standard can be easily accessed at random points. Thus, a presentation of guidelines for content authors regarding random access can encourage the development of randomly-accessible bitstreams when this functionality is needed for a particular application. The techniques here are presented for informative purposes only, and are not required for compliance to this part of ISO/IEC 14496.

5.E.2 Difficulties in general-purpose random access

The major problem with enabling random access to every bitstream is that of hidden state. Consider the following orchestra:

```
global {
  srate 24000;
  krate 1000;
  ivar count;
}

instr alternate() {
  imports exports ivar count;
  count = count + 1;
  if ((floor(count/2) * 2 == count && ltime == 0)  {  // divisible by 2?
    instr tone(...);
  }
  // notice no regular output
}

instr tone(...) { ... }  // make some sound
```

This orchestra counts up the number of times the alternate() instrument is instantiated. Every even-numbered instance, it passes on control to the tone() instrument (which presumably makes some noise); on the odd-numbered instances, nothing happens. Thus, with a score containing the following subclause:

```
.
.
.
25.0 alternate 1.0
26.0 alternate 1.0
27.0 alternate 1.0
.
.
.
```

it is impossible to determine from this segment alone the parity of the first instrument line referring to alternate(); we cannot randomly access this point in the sound except by scanning backward through the score to discover how many previous times the instrument has been instantiated.

The general problem here is with the *hidden state variable* count. The instrument alternate() depends on **count** for its behavior, and this variable has two properties: (1) it has a long “memory” of its past behavior (it maintains state), and (2) it is not controlled by the score (it is hidden). Any orchestra that depends on such hidden state variables is not easily randomly-accessible.

The example here generalizes, not only to instrument dispatch, but to controllers, tempo changes, and even to the development of very long notes containing ambient sound. Suppose, for instance, that there was a long note begun in the above score at time 2.0 that is supposed to dominate the sound presentation at time 25.0 – in order to discover this, we have to similarly scan backward through the score.

5.E.3 Making Structured Audio bitstreams randomly-accessible

5.E.3.1 Introduction

This Annex describes recommended practices for providing random access points to Structured Audio bitstreams. It covers two topics. First, a set of constructs to avoid is presented; then, a discussion is conducted regarding methods for converting non-randomly-accessible to randomly-accessible bitstreams.

5.E.3.2 Constructs to avoid

In order to make bitstreams randomly-accessible, the following constructs should only be used carefully. Note that there is nothing “wrong” with this constructs in general, and they are useful in applications where random access to sound data is not required. Note also that examples can be constructed for each of these in which the described technique is used and random-accessibility is still maintained.

1. Very long notes. A note that lasts for a long time tends to prohibit random access while it is sustaining, particularly if the sound it makes changes a great deal over the duration.

2. Global variables. These are often used to preserve state between instrument instances or to share data between instruments. When used in this way, they tend to add hidden state to the orchestra.

3. Score-based creation of wavetables. If random access skips over the important point at which a wavetable is created, then it will not be available if some instrument instance needs it.

4. Modification of global wavetables. This is a special case of point (2).

5. Complex score-based control. If score-based controllers are used to modify greatly the behaviour of the orchestra, then random access will produce the wrong result if the control instructions are skipped. This applies to MIDI controllers as well as SASL controllers.

If these constructs are avoided, then in general a bitstream allows random access at the point of each instrument line in the score.

5.E.3.3 Altering bitstreams to make them randomly accessible

5.E.3.3.1 Introduction

For each of the points listed above, it is possible to modify a bitstream that uses the technique to create a bitstream that produces the same sound, but is higher-bandwidth and randomly accessible. This annex provides general guidelines for accomplishing this. Note that it is not possible (due to computational incompleteness) to develop an automated tool that determines whether a particular bitstream is randomly accessible, but it is possible to develop a tool that applies transforms of the following sort automatically.

5.E.3.3.2 Break up very long notes

Very long notes can be broken into several shorter notes. To take a simple example:

```
global {
  krate 100;
}

instr tone(freq) {
```

```
  asig a;
  table sine(harm,2048,1);
  a = oscil(sine,freq);
  output(a);
}

tone 10.0 440
```

This orchestra and score play a 440-Hz sine tone for 10 seconds. The same sound is produced by the score

```
0.0 tone 0.99 440
1.0 tone 0.99 440
2.0 tone 0.99 440
3.0 tone 0.99 440
4.0 tone 0.99 440
5.0 tone 0.99 440
6.0 tone 0.99 440
7.0 tone 0.99 440
8.0 tone 0.99 440
9.0 tone 0.99 440
```

which is randomly-accessible once per second rather than only at the beginning. Note that the sound is the same here because the tone is only broken into a new note at the zero-phase point so that the phase remains continuous. (The duration is 0.99 because note instances execute for one extra control period, as specified in subclause 5.7.3.3.6).

For a note that changes an internal state during playback, the internal state must be exposed as a p-field (or control parameter). For example:

```
instr tone(freq) {
  ksig env;
  asig a;
  table sine(harm,2048,1);
  env = kline(0,dur/2,1,dur/2,0);
  a = oscil(sine,freq) * env;
  output(a);
}

0.0 tone 10.0 440
```

In this case, the **kline()** core opcode encapsulates hidden state (the “current value” of the line segment) that must be exposed to break the note into sections. Thus:

```
instr tone(freq, startenv, endenv) {
  ksig env;
  asig a;
  table sine(harm,2048,1);
  env = kline(startenv,dur,endenv);
  a = oscil(sine,freq) * env;
  output(a);
}

0.0 tone 0.99 440 0.0 0.2
1.0 tone 0.99 440 0.2 0.4
2.0 tone 0.99 440 0.4 0.6
3.0 tone 0.99 440 0.6 0.8
4.0 tone 0.99 440 0.8 1.0
5.0 tone 0.99 440 1.0 0.8
6.0 tone 0.99 440 0.8 0.6
7.0 tone 0.99 440 0.6 0.4
8.0 tone 0.99 440 0.4 0.2
9.0 tone 0.99 440 0.2 0.0
```

This orchestra/score specifies the same sound, but is randomly accessible once per second. Application of this technique clearly becomes difficult for very complex instruments.

5.E.3.3.3 Replace global variables with controllers

Consider the initial example from subclause 5.E.2:

```
global {
  srate 24000;
  krate 1000;
  ivar count;
}

instr alternate() {
  imports exports ivar count;
  count = count + 1;
  if ((floor(count/2) * 2 == count && ltime == 0)  {  // divisible by 2?
    instr tone(...);
  }
  // notice no regular output
}

instr tone(...) { ... }  // make some sound

.
.
.
25.0 alternate 1.0
26.0 alternate 1.0
27.0 alternate 1.0
.
.
.
```

We can make this bitstream randomly accessible by moving the manipulation of the **count** global variable out of the **alternate** instrument and into the score.

```
global {
  srate 24000;
  krate 1000;
  imports ivar count;
}

instr alternate() {
  imports ivar count;

  if ((floor(count/2) * 2 == count && ltime == 0)  {  // divisible by 2?
    instr tone(...);
  }
  // notice no regular output
}

instr tone(...) { ... }  // make some sound

.
.
25.0 control count 14.0
25.0 alternate 1.0
26.0 control count 15.0
26.0 alternate 1.0
27.0 control count 16.0
27.0 alternate 1.0
.
.
.
```

With this change, the bitstream is now randomly accessible (and requires twice the bandwidth).

5.E.3.3.4 Replicate score-based wavetable generators

Consider the following example:

```
global {
  srate 24000;
  krate 1000;
}

instr tone(freq) {
  imports table shape;

  output(oscil(shape,freq));
}

0.0 table shape harm 2048 1
.
.
.
25.0 tone 1.0 440 // *
26.0 tone 1.0 485
```

At the random-access point (marked *), it is not possible to determine the desired waveshape from local information. To fix this, we simply replicate the table generator throughout the score:

```
0.0 table shape harm 2048 1
.
.
.
25.0 table shape harm 2048 1
25.0 tone 1.0 440 // *
26.0 table shape harm 2048 1
26.0 tone 1.0 485
```

This bitstream is again more expensive, both in bandwidth and in computation, but it allows random access.

Annex 5.F
(informative)

Directly-connected MIDI and microphone control of the orchestra

5.F.1 Introduction

Although it is outside the normative scope of this part of ISO/IEC 14496, some discussion of connecting live MIDI devices and other controllers, and live microphones, directly to a sound generator making use of the Structured Audio tools (especially a SAOL orchestra) is presented and recommended practices developed. This discussion is presented for informative purposes only; the techniques presented here are not required for compliance to this part of ISO/IEC 14496.

The connection of live MIDI control of an orchestra allows a Structured Audio decoder device to be used as a real-time musical instrument in performance or recording situations. The sound quality and flexibility of the Structured Audio tools is an improvement over fixed hardware synthesizers for use in these situations. The connection of live microphones allows a Structured Audio decoder to act as an effects processor, a live or interactive karaoke device, or to enable other kinds of interactive electroacoustic performances.

This Annex presents recommendations for the connection of live MIDI devices and other controllers, and live microphones, to a Structured Audio decoder.

5.F.2 MIDI controller recommended practices

As the MIDI-event bitstream format is very similar to the MIDI control data generated by a live MIDI device (it exactly encapsulates the data bytes generated by such a device in the MPEG-4 Access Unit), no direct modification needs to be made to a Structured Audio decoder to enable control of an orchestra by such a devices. All that need be accomplished is a connection between the MIDI input of the terminal and the scheduler input, so that non-timestamped events from the MIDI input are passed directly to the scheduler. The following practices are recommended in this method:

The MIDI input should not be converted into a legal Structured Audio bitstream, but should generate events directly in the scheduler.

Any note-on events generated with a live MIDI device should not be executed in the order prescribed by the global sequencing rules (see subclause 5.8.5.6). Rather, the note instantiation and first k-cycle of the instrument instance should be executed in the current orchestra pass as soon as possible after they are received by the orchestra. Upon the second k-cycle pass through the instrument instance, the instance begins to be processed according to the global sequencing rules. This practice may cause unpredictable results if it is used in conjunction with instruments containing certain global-variable or bus-routing constructions, and thus such constructions must be used advisedly. If possible, the latency between the time the event is triggered by the live performer and the time at which the first k-cycle of the instrument sound is audible should be no greater than 5 ms.

The live MIDI events should not be subject to orchestra tempo control.

The live MIDI events should otherwise be treated as any other orchestra MIDI event. Streaming performance and live performance should be possible at the same time.

If multiple MIDI devices are connected to the same terminal, the terminal should allow that the channel numbering be managed logically, so that “MIDI channel 1” from Device A is a different channel than “MIDI channel 1” from Device B. For this purpose, it is recommended that each such MIDI device be assigned device numbers in ascending monotonic sequence. The **channel** value for a particular live MIDI event is then given by:

channel = live MIDI event channel + (device number * 16).

The initial device number should be chosen so that **channel** values assigned to live MIDI events do not conflict with **channel** values assigned to **midi_file** events as described in subclause 5.14.3.3.

Although at the time of writing ISO/IEC 14496, the vast majority of live musical control devices generate MIDI data, it is also possible to construct musical control devices that generate SASL events directly. In this case, the recommended practices are the same as above, except the incoming data is represented in terms of non-timestamped SASL events. The method of constructing devices that generate legal SASL events in real-time is outside the scope of this Annex.

5.F.3 Live microphone recommended practices

As the Structured Audio orchestra already handles acoustic data very well, it is easy to allow the connection of a live microphone. The special bus **input_bus** is defined as in subclause 5.15.2, and the audio data captured by the microphone is placed on this bus, from which it can be sent to other instruments for processing. The following practices are recommended in this method:

A special bus called **input_bus** is defined. This bus has the number of channels specified by the **inchannels** global parameter (see subclause 5.8.5.2.3). If the **inchannels** global parameter is not specified, this bus (and thus the microphone) cannot be used.

At the beginning of each control cycle (i.e., between step 9 and step 10 of subclause 5.7.3.3.6), the microphone input is sampled at the orchestra sampling rate and placed on the special bus **input_bus**. If there are more channels of microphone input than on **input_bus**, only the first channels are used and the rest are discarded; if there are fewer, then the “extra” channels of **input_bus** are set to all 0 values. If there is no microphone connected, then all channels of **input_bus** are set to all 0 values.

The **input_bus** is treated as any other bus in the orchestra.

There is no recommended practice for using a microphone in conjunction with a SAOL orchestra that is used to process effects for an AudioFX node.

Bibliography

[BIFS] Scheirer, Eric D., and Riitta Väänänen and Jyri Huopaniemi, "AudioBIFS: The MPEG-4 standard for effects processing." *Proc. 1st Cost-G6 Workshop on Digital Audio Effects Processing (DAFX-98)*, Barcelona, 1998.

[BOOK] Scheirer, Eric D., and Youngjik Lee and Jae-Woo Yang, "Synthetic audio and SNHC audio in MPEG-4." In Puri, Atul and Tsuhan Chen (eds.), *Advances in Multimedia: Systems, Standards, and Networks*. New York: Marcel Dekker, in press.

[DRAG] Aho, Alfred V., and Ravi Sethi and Jeffrey Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, Mass: Addison-Wesley, 1984.

[GRAN] Cavaliere, Sergio, and Aldo Piccialli, "Granular synthesis of musical signals", in Curtis Roads, Stephen Travis Pope, Aldo Piccialli, and Giovanni de Poli (eds.), *Musical Signal Processing*. London: Swets & Zeitlinger, 1998, pp. 155-186.

[ICASSP] Scheirer, Eric D., "The MPEG-4 Structured Audio standard", *Proc 1998 IEEE ICASSP*, Seattle, 1998, pp. 3801-3804.

[NETSOUND] Casey, Michael A., and Paris G. Smaragdis, "Netsound", *Proc. 1996 ICMC*, Hong Kong, 1996, p. 143

[SAFX] Scheirer, Eric D., "Structured audio and effects processing in the MPEG-4 multimedia standard", *Multimedia Systems* **7**:1 (Jan. 1999), pp. 11-22.

[SAOL] Scheirer, Eric D., and Barry L. Vercoe, "SAOL: The MPEG-4 Structured Audio Orchestra Language", *Computer Music Journal,* in press.

[SAUD] Vercoe, Barry, and William G. Gardner and Eric D. Scheirer , "Structured Audio: Creation, Transmission, and Rendering of Parametric Sound Descriptions". *Proc. IEEE* **85**:5 (May 1998), pp. 922-940.

[WAVE] Scheirer, Eric D., and Lee Ray, "Algorithmic and wavetable synthesis in the MPEG-4 multimedia standard". Presented at the 105th Convention of the Audio Engineering Society, San Francisco, 1998 (AES reprint #4811).

Alphabetical Index to Subpart 5 of ISO/IEC 14496-3

Page numbers in **boldface** refer to definitions in subclause 5.3.

Contents for Subpart 6

Subpart 6: Text to Speech Interface (TTSI)

6.1 Scope

This subpart of ISO/IEC 14496-3 specifies the coded representation of MPEG-4 Audio Text-to-Speech (M-TTS) and its decoder for high quality synthesized speech and for enabling various applications. The exact synthesis method is not a standardization issue partly because there are already various speech synthesis techniques.

This subpart of ISO/IEC 14496-3 is intended for application to M-TTS functionalities such as those for facial animation (FA) and moving picture (MP) interoperability with a coded bitstream. The M-TTS functionalities include a capability of utilizing prosodic information extracted from natural speech. They also include the applications to the speaking device for FA tools and a dubbing device for moving pictures by utilizing lip shape and input text information.

The text-to-speech (TTS) synthesis technology is recently becoming a rather common interface tool and begins to play an important role in various multimedia application areas. For instance, by using TTS synthesis functionality, multimedia contents with narration can be easily composed without recording natural speech sound. Moreover, TTS synthesis with facial animation (FA) / moving picture (MP) functionalities would possibly make the contents much richer. In other words, TTS technology can be used as a speech output device for FA tools and can also be used for MP dubbing with lip shape information. In MPEG-4, common interfaces only for the TTS synthesizer and for FA/MP interoperability are defined. The M-TTS functionalities can be considered as a superset of the conventional TTS framework. This TTS synthesizer can also utilize prosodic information of natural speech in addition to input text and can generate much higher quality synthetic speech. The interface bitstream format is strongly user-friendly: if some parameters of the prosodic information are not available, the missed parameters are generated by utilizing preestablished rules. The functionalities of the M-TTS thus range from conventional TTS synthesis function to natural speech coding and its application areas, i.e., from a simple TTS synthesis function to those for FA and MP.

6.2 Definitions

Definitions can be found in subpart 1, subclause 1.3.

6.3 Symbols and abbreations

F0 fundamental frequency (pitch frequency)

DEMUX demultiplexer

FA facial animation

FAP facial animation parameter

ID identifier

IPA International Phonetic Alphabet

MP moving picture

M-TTS MPEG-4 Audio TTS

STOD story teller on demand

TTS text-to-speech

6.4 MPEG-4 audio text-to-speech bitstream syntax

6.4.1 MPEG-4 audio TTSSpecificConfig

TTSSpecificConfig() {

TTS_Sequence()

}

Table 6.1 — Syntax of TTS_Sequence()

Syntax	No. of bits	Mnemonic
TTS_Sequence()		
{		
TTS_Sequence_ID;	**5**	**uimsbf**
Language_Code;	**18**	**uimsbf**
Gender_Enable;	**1**	**bslbf**
Age_Enable;	**1**	**bslbf**
Speech_Rate_Enable;	**1**	**bslbf**
Prosody_Enable;	**1**	**bslbf**
Video_Enable;	**1**	**bslbf**
Lip_Shape_Enable;	**1**	**Bslbf**
Trick_Mode_Enable;	**1**	**Bslbf**
}		

6.4.2 MPEG-4 audio text-to-speech payload

AlPduPayload {

TTS_Sentence();

}

Table 6.2 — Syntax of TTS_Sentence()

Syntax	**No. of bits**	**Mnemonic**
TTS_Sentence() {		
TTS_Sentence_ID;	**10**	**uimsbf**
Silence;	**1**	**bslbf**
if (Silence) {		
Silence_Duration;	**12**	**uimsbf**
}		
else {		
if (Gender_Enable) {		
Gender;	**1**	**bslbf**
}		
if (Age_Enable) {		
Age;	**3**	**uimsbf**
}		
if (!Video_Enable && Speech_Rate_Enable) {		
Speech_Rate;	**4**	**uimsbf**
}		
Length_of_Text;	**12**	**uimsbf**
for (j = 0; j < Length_of_Text; j++) {		
TTS_Text;	**8**	**bslbf**
}		
if (Prosody_Enable) {		

```
                Dur_Enable;                                          1          bslbf
                F0_Contour_Enable;                                   1          bslbf
                Energy_Contour_Enable;                               1          bslbf
                Number_of_Phonemes;                                  10         uimsbf
                Phoneme_Symbols_Length;                              13         uimsbf
                for (j = 0 ; j < Phoneme_Symbols_Length ; j++) {
                    Phoneme_Symbols;                                 8          bslbf
                }
                for (j = 0 ; j < Number_of_Phonemes ; j++) {
                    if (Dur_Enable) {
                        Dur_each_Phoneme;                            12         uimsbf
                    }
                    if (F0_Contour_Enable) {
                        Num_F0;                                      5          uimsbf
                        for (k = 0; k < Num_F0;k++) {
                            F0_Contour_each_Phoneme;                 8          uimsbf
                            F0_Contour_each_Phoneme_Time;            12         uimsbf
                        }
                    }
                    if (Energy_Contour_Enable) {
                        Energy_Contour_each_Phoneme;                 8*3=24     uimsbf
                    }
                }
            }
            if (Video_Enable) {
                Sentence_Duration;                                   16         uimsbf
                Position_in_Sentence;                                16         uimsbf
                Offset;                                              10         uimsbf
            }
            if (Lip_Shape_Enable) {
                Number_of_Lip_Shape;                                 10         uimsbf
                for (j = 0 ; j < Number_of_Lip_Shape ; j++) {
                    Lip_Shape_in_Sentence;                           16         uimsbf
                    Lip_Shape;                                       8          uimsbf
                }
            }
        }
}
```

6.5 MPEG-4 audio text-to-speech bitstream semantics

6.5.1 MPEG-4 audio TTSSpecificConfig

TTS_Sequence_ID This is a five-bit ID to uniquely identify each TTS object appearing in one scene. Each speaker in a scene will have distinct TTS_Sequence_ID.

Language_Code When this is "00" (00110000 00110000 in binary), the IPA is to be sent. In all other languages, this is the ISO 639 Language Code. In addition to this 16 bits, two bits that represent dialects of each language is added at the end (user defined).

Gender_Enable This is a one-bit flag which is set to '1' when the gender information exists.

Age_Enable This is a one-bit flag which is set to '1' when the age information exists.

Speech_Rate_Enable This is a one-bit flag which is set to '1' when the speech rate information exists.

Prosody_Enable This is a one-bit flag which is set to '1' when the prosody information exists.

Video_Enable This is a one-bit flag which is set to '1' when the M-TTS decoder works with MP. In this case, M-TTS should synchronize synthetic speech to MP and accommodate the functionality of ttsForward and ttsBackward. When VideoEnable flag is set, M-TTS decoder uses system clock to select adequate TTS_Sentence frame and fetches Sentence_Duration, Position_in_Sentence, Offset data. TTS synthesizer assigns appropriate duration for each phoneme to meet Sentence_Duration. The starting point of speech in a sentence is decided by Position_in_Sentence. If Position_in_Sentence equals 0 (the starting point is the initial of sentence), TTS uses Offset as a delay time to synchronize synthetic speech to MP.

Lip_Shape_Enable This is a one-bit flag which is set to '1' when the coded input bitstream has lip shape information. With lip shape information, M-TTS request FA tool to change lip shape according to timing information (Lip_Shape_in_Sentence) and predifined lip shape pattern.

Trick_Mode_Enable This is a one-bit flag which is set to '1' when the coded input bitstream permits trick mode functions such as stop, play, forward, and backward.

6.5.2 MPEG-4 audio text-to-speech payload

TTS_Sentence_ID This is a ten-bit ID to uniquely identify a sentence in the M-TTS text data sequence for indexing purpose. The first five bits equal to the TTS_Sequence_ID of the speaker defined in subclause 6.5.1, and the rest five bits are the sequential sentence number of each TTS object.

Silence This is a one-bit flag which is set to '1' when the current position is silence.

Silence_Duration This defines the time duration of the current silence segment in milliseconds. It has a value from 1 to 4095. The value '0' is prohibited.

Gender This is a one-bit which is set to '1' if the gender of the synthetic speech producer is male and '0', if female.

Age This represents the age of the speaker for synthetic speech. The meaning of age is defined in Table 6.3.

Table 6.3 — Age mapping table

Age	Age of the speaker
000	below 6
001	6 – 12
010	13 – 18
011	19 – 25
100	26 – 34
101	35 – 45
110	45 – 60
111	over 60

Speech_Rate This defines the synthetic speech rate in 16 levels. The level 8 corresponds the normal speed of the speaker defined in the current speech synthesizer, the level 0 corresponds to the slowest speed of the speech synthesizer, and the level 15 corresponds to the fastest speed of the speech synthesizer.

Length_of_Text This identifies the length of the TTS_Text data in bytes.

TTS_Text This is a character string containing the input text. The text bracketed by < and > contains bookmarks. If the text bracketed by < and > starts with FAP, the bookmark is handed to the face animation through the TtsFAPInterface as a string of characters. Otherwise, the text of the bookmark is ignored. The syntax of the bookmarks is defined in ISO/IEC 14496-2.

Dur_Enable This is a one-bit flag which is set to '1' when the duration information for each phoneme exists.

F0_Contour_Enable This is a one-bit flag which is set to '1' when the pitch contour information for each phoneme exists.

Energy_Contour_Enable This is a one-bit flag which is set to '1' when the energy contour information for each phoneme exists.

Number_of_Phonemes This defines the number of phonemes needed for speech synthesis of the input text.

Phonemes_Symbols_Length This identifies the length of Phonemes_Symbols (IPA code) data in bytes since the IPA code has optional modifiers and dialect codes.

Phoneme_Symbols This defines the indexing number for the current phoneme by using the Unicode 2.0 numbering system. Each phoneme symbol is represented as a number for the corresponding IPA. Three two-byte numbers is used for each IPA representation including a two-byte integer for the character, and an optional two-byte integer for the spacing modifier, and another optional two-byte integer for the diacritical mark.

Dur_each_Phoneme This defines the duration of each phoneme in msec.

Num_F0 This defines the number of F0 values specified for the current phoneme.

F0_Contour_each_Phoneme This defines half of the F0 value in Hz at time instant F0_Contour_each_Phoneme_Time.

F0_Contour_each_Phoneme_Time This defines the integer number of the time in ms for the position of the F0_Contour_each_Phoneme.

Energy_Contour_each_Phoneme These 3 8-bit data correspond to the energy values at the start, the middle, and the end positions of the phoneme. The energy value X is calculated as

$$X = \mathrm{int}(50 \log_{10} A_{p-p}),$$

where A_{p-p} is the peak-to-peak value of the speech waveform at the defined position.

Sentence_Duration This defines the duration of the sentence in msec.

Position_in_Sentence This defines the position of the current stop in a sentence as an elapsed time in msec.

Offset This defines the duration of a very short pause before the start of synthesized speech output in msec.

Number_of_Lip_Shape This defines the number of lip-shape patterns to be processed.

Lip_Shape_in_Sentence This defines the position of each lip shape from the beginning of the sentence in msec.

Lip_Shape This defines the indexing number for the current lip-shape pattern to be processed that is defined in ISO/IEC 14496-2.

6.6 MPEG-4 audio text-to-speech decoding process

The architecture of the M-TTS decoder is described below and only the interfaces relevant to the M-TTS decoder are the subjects of standardization. The number above each arrow indicates the section describing each interface.

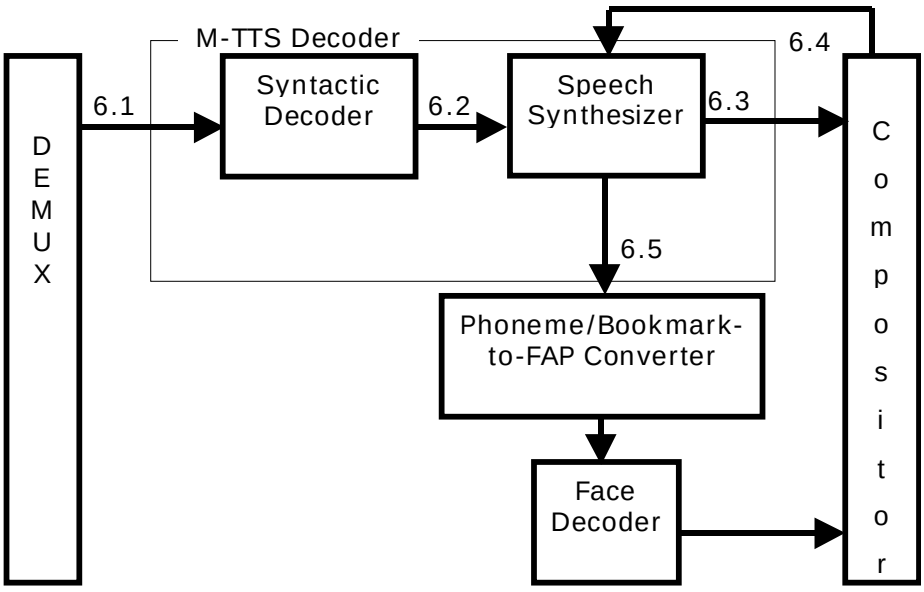


Figure 6.1 — MPEG-4 Audio TTS decoder architecture.

In this architecture the following types of interfaces is distinguished:

Interface between DEMUX and the syntactic decoder

Interface between the syntactic decoder and the speech synthesizer

Interface from the speech synthesizer to the compositor

Interface from the compositor to the speech synthesizer

Interface between the speech synthesizer and the phoneme/bookmark-to-FAP converter

6.6.1 Interface between DEMUX and syntactic decoder

Receiving a bitstream, DEMUX passes coded M-TTS bitstreams to the syntactic decoder.

6.6.2 Interface between syntactic decoder and speech synthesizer

Receiving a coded M-TTS bitstream, the syntactic decoder passes some of the following bitstreams to the speech synthesizer.

Input type of the M-TTS data: specifies synchronized operation with FA or MP

Control commands stream: Control command sequence

Input text: character string(s) for the text to be synthesized

Auxiliary information: Prosodic parameters including phoneme symbols

Lip shape patterns

Information for trick mode operation

The pseudo-C code representation of this interface is defined in subclause 6.4.

6.6.3 Interface from speech synthesizer to compositor

This interface is identical to the interface for digitized natural speech to the compositor. The dynamic range is from – 32767 to + 32768.

6.6.4 Interface from compositor to speech synthesizer

This interface is defined to allow the local control of the synthesized speech by users. This user interface supports trick mode of the synthesized speech in synchronization with MP and changes some prosodic properties of the synthesized speech by using the ttsControl defined as follows:

Table 6.4 — Syntax of ttsControl()

Syntax	No. of bits	Mnemonic

ttsControl()
{
ttsPlay();
ttsForward();
ttsBackward();
ttsStopSyllable();
ttsStopWord();
ttsStopPhrase();
TtsChangeSpeedRate();
TtsChangePitchDynamicRange();
TtsChangePitchHeight();
TtsChangeGender();
ttsChangeAge();
}		

The member function ttsPlay allows a user to start speech synthesis in the forward direction while ttsForward and ttsBackword enable the user to change the starting play position in forward and backward direction, respectively. The ttsStopSyllable, ttsStopWord, and ttsStopPhrase functions define the interface for users to stop speech synthesis at the specified boundary such as syllable, word, and phrase. The member function ttsChangeSpeechRate is an interface to change the synthesized speech rate. The argument speed has the numbers from 1 to 16. The member function ttsChangePitchDynamicRange is an interface to change the dynamic range of the pitch of synthesized speech. By using the argument of this function, level, a user can change the dynamic range from 1 to 16. Also a user can change the pitch height from 1 to 16 by using the argument height in the member function ttsChangePitchHeight. The member functions ttsChangeGender and ttsChangeAge allow a user to change the gender and the age of the synthetic speech producer by assigning numbers, as defined in subclause 6.5.2, to their arguments, gender and age, respectively.

6.6.5 Interface between speech synthesizer and phoneme/bookmark-to-FAP converter

In the MPEG-4 framework, the speech synthesizer and the face animation are driven synchronously. The speech synthesizer generates synthetic speech. At the same time, TTS gives phonemeSymbol and phonemeDuration as well as bookmarks to the Phoneme/Bookmark-to-FAP converter. The Phoneme/Bookmark to FAP converter generates relevant facial animation according to the phonemeSymbol, the phonemeDuration and bookmarks. Further description of the Phoneme/Bookmark to FAP converter is provided in ISO/IEC 14496-2.

The synthesized speech and facial animation have relative synchronization except the absolute composition time. The synchronization of the absolute composition time comes from the same composition time stamp of the TTS bitstream. If the Lip_Shape_Enable is set, the Lip_Shape_in_Sentence is used to generate the phonemeDuration. Otherwise, the TTS provides phoneme durations. The speech synthesizer generates stress and/or wordBegin bits when the corresponding phoneme has stress and/or start of a word, respectively.

Within the MTTS_Text, the beginning of a bookmark for using facial animation parameters is identified by '<FAP'. The bookmark lasts until the closing bracket '>'

A bookmark is handed to the TtsFAPInterface with the phoneme of the next word of the current sentence following the bookmark. If there is no word after the bookmark, the bookmark is handed to the TtsFAPInterface with the last phoneme of the previous word in the current sentence. In order to allow animation of complex expressions and motion, a sequence of up to 40 bookmarks is allowed without words between them. The starttime defines the time in msec relative to the beginning of the M-TTS sequence when the phoneme will start playing.

The class ttsFAPInterface defines the data structure for the interface between the speech synthesizer and the phoneme-to-FAP converter.

Table 6.5 — Syntax of TtsFAPInterface()

Syntax	No. of bits	Mnemonic
TtsFAPInterface()		
{		
PhonemeSymbol;	**8**	**uimsbf**
PhonemeDuration;	**12**	**uimsbf**
f0Average;	**8**	**uimsbf**
Stress;	**1**	**bslbf**
WordBegin;	**1**	**bslbf**
Bookmark;		**char ***
Starttime;		**long int**
}		

Annex 6.A
(informative)

Applications of MPEG-4 audio text-to-speech decoder

6.A.1 General

This annex part describes application scenarios for the M-TTS decoder.

6.A.2 Application scenario: MPEG-4 Story Teller on Demand (STOD)

In the STOD application, users can select a story from a huge database of story libraries which are stored in hard disks or compact disks. The STOD system reads aloud the story via the M-TTS decoder with the MPEG-4 facial animation tool or with appropriately selected images. The user can stop and resume speaking at any moment he wants through user interfaces of the local machine (for example, mouse or keyboard). The user can also select the gender, age, and the speech rate of the electronic story teller.

The synchronization between the M-TTS decoder with the MPEG-4 facial animation tool is realized by using the same composition time of the M-TTS decoder for the MPEG-4 facial animation tool.

6.A.3 Application scenario: MPEG-4 audio text-to-speech with moving picture

In this application, synchronized playback of the M-TTS decoder and encoded moving picture is the most important issue. The architecture of the M-TTS decoder can provide several granularities of synchronization. Aligning the composition time of each TTS_Sentence, coarse granularity of synchronization and trick mode functionality can be easily achieved. To get finer granularity of synchronization, the information about the Lip_Shape would be utilized. The finest granularity of synchronization can be achieved by using the prosody information and the video-related information such as Sentence_Duration, Position_in_Sentence, and Offset.

With this synchronization capability, the M-TTS decoder can be used for moving picture dubbing by utilizing the Lip_Shape and Lip_Shape_in_Sentence.

6.A.4 MPEG-4 audio TTS and face animation using bookmarks appropriate for trick mode

Bookmarks allow to animate a face using facial animation parameters (FAP) in addition to the animation of the mouth derived from phonemes. The FAP of the bookmark is applied to the face until another bookmark resets the FAP. Designing contents that replay each sentence independent of trick mode requires that bookmarks of the text to be spoken are repeated at the beginning of each sentence to initialize the face to the state that is defined by the previous sentence. In this case, some mismatch of synchronization can occur in the beginning of a sentence. However, the system recovers when the new bookmark is processed.

6.A.5 Random access unit

Every TTS_Sentence is a random access unit.

Contents for Subpart 7

Subpart 7: Parametric Audio Coding - HILN

7.1 Scope

The Parametric Audio Coding Subpart provides the HILN tools which complement the other natural audio coding tools in the area of very low bit rates. Their focus is the representation of monophonic music signals with low and intermediate content complexity in the range of 4 to 16 kbit/s. HILN enables a high grade of interactivity by implicit support of speed and pitch change during playback and by the capability of bit rate scalability. Furthermore the possible combination with the parametric speech coding tools HVXC allows very efficient schemes for coding speech and music signals.

7.1.1 Technical Overview

MPEG-4 parametric audio coding uses the HILN technique (Harmonic and Individual Lines plus Noise) to code audio signals like music at bitrates of 4 kbit/s and higher using a scalable parametric representation of the audio signal. HILN allows independent change of speed and pitch during decoding. Furthermore HILN can be combined with MPEG-4 parametric speech coding (HVXC, see Subpart 2) to form an integrated parametric coder covering a wider range of signals and bitrates.

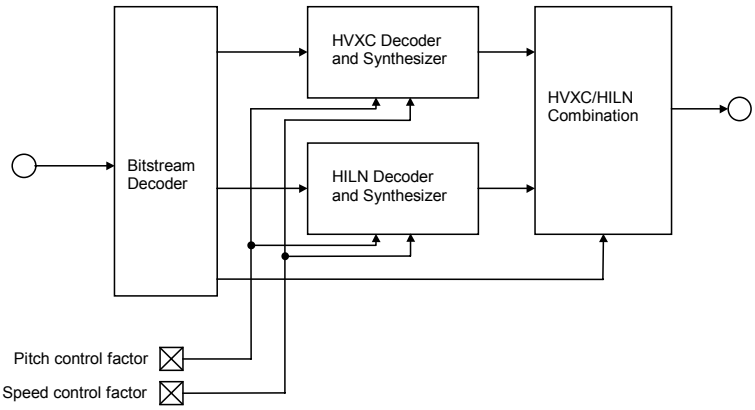


Figure 7.1 — Block diagram of the integrated parametric decoder

The integrated parametric coder can operate in the following modes:

Table 7.1 — Parametric coder operation modes

PARAmode	Description
0	HVXC only
1	HILN only
2	switched HVXC / HILN
3	mixed HVXC / HILN

PARAmodes 0 and 1 represent the fixed HVXC and HILN modes. PARAmode 2 permits automatic switching between HVXC and HILN depending on the current input signal type. In PARAmode 3 the HVXC and HILN decoders can be used simultaneously and their output signals are added (mixed) in the parametric decoder.

In “switched HVXC / HILN” and “mixed HVXC / HILN” modes both HVXC and HILN decoder tools are operated alternatively or simultaneously according to the PARAswitchMode or PARAmixMode of the current frame. To obtain proper time alignment of both HVXC and HILN decoder output signals before they are added, a FIFO buffer compensates for the time difference between HVXC and HILN decoder delay.

To avoid hard transitions at frame boundaries when the HVXC or HILN decoders are switched on or off, the respective decoder output signals fade in and out smoothly. For the HVXC decoder a 20 ms linear fade is applied when it is switched on or off. The HILN decoder requires no additional fading because of the smooth synthesis windows utilized in the HILN synthesizer. It is only necessary to reset the HILN decoder (numLine = 0) if the current bitstream frame contains no HILNframe().

7.2 Definitions

Definitions can be found in subpart 1, subclause 1.3.

7.3 Bitstream syntax

An MPEG-4 Natural Audio Object using Parametric Coding is transmitted in one or more Elementary Streams: The base layer stream, an optional enhancement layer stream, and one or more optional extension layer streams.

The bitstream syntax is described in pseudo-C code.

The mnemonics LARH1, LARH2, LARH3, LARN1, LARN2, DIA, DIF, DHF, DFS indicate that a “vlclbf” codeword is used. The corresponding codebooks are given in subclause 7.3.2.3.

The mnemonic SDC indicates that a “vlclbf” codeword is used which is decoded by the HILN SubDivisionCode described in subclause 7.3.2.4 using the parameters for SDCdecode() as given in the bitstream syntax description.

7.3.1 Decoder configuration (ParametricSpecificConfig)

The decoder configuration information for parametric coding is transmitted in the ParametricSpecificConfig() of the base layer and the enhancement or extension layer Elementary Stream (see Subpart 1).

Parametric Base Layer -- Configuration
The parametric coder in unscalable mode or the base layer in HILN scalable mode uses a ParametricSpecificConfig() with isBaseLayer == 1.

Parametric HILN Enhancement / Extension Layer -- Configuration
To use HILN as core in a "T/F scalable with core" mode, in addition to the HILN base layer an HILN enhancement layer is required. In HILN bitrate scalable operation, in addition to the HILN base layer one or more HILN extension layers are permitted. Both the enhancement and extension layer use a ParametricSpecificConfig() with isBaseLayer == 0.

Table 7.2 — Syntax of ParametricSpecificConfig()

Syntax	No. of bits	Mnemonic
ParametricSpecificConfig()		
{		
isBaseLayer;	**1**	**uimsbf**
if (isBaseLayer) {		
PARAconfig();		
}		
else {		
HILNenexConfig();		
}		
}		

7.3.1.1 Parametric Audio decoder configuration

Table 7.3 — Syntax of PARAconfig()

Syntax	No. of bits	Mnemonic
PARAconfig()		
{		
PARAmode;	**2**	**uimsbf**
if (PARAmode != 1) {		
ErHVXCconfig();		
}		
if (PARAmode != 0) {		
HILNconfig();		
}		
PARAextensionFlag;	**1**	**uimsbf**
if (PARAextensionFlag) {		
/* to be defined in MPEG-4 Phase 3 */		
}		
}		

Table 7.4 — PARAmode

PARAmode	frameLength	Description
0	20 ms (N = 160 samples)	HVXC only
1	see subclause 7.3.1.2 and subclause 7.5.1.4.3.3	HILN only
2	40 ms (N = 320 samples)	HVXC/HILN switching
3	40 ms (N = 320 samples)	HVXC/HILN mixing

7.3.1.2 HILN decoder configuration

Table 7.5 — Syntax of HILNconfig()

Syntax	No. of bits	Mnemonic
HILNconfig()		
{		
HILNquantMode;	**1**	**uimsbf**
HILNmaxNumLine;	**8**	**uimsbf**
HILNsampleRateCode;	**4**	**uimsbf**
HILNframeLength;	**12**	**uimsbf**
HILNcontMode;	**2**	**uimsbf**
}		

Table 7.6 — Syntax of HILNenexConfig()

Syntax	No. of bits	Mnemonic
HILNenexConfig()		
{		
HILNenhaLayer;	**1**	**uimsbf**
if (HILNenhaLayer) {		
HILNenhaQuantMode;	**2**	**uimsbf**
}		
}		

Table 7.7 — HILNsampleRateCode

HILNsampleRateCode	sampleRate	maxFIndex
0	96000	890
1	88200	876
2	64000	825
3	48000	779
4	44100	765
5	32000	714
6	24000	668
7	22050	654
8	16000	603
9	12000	557
10	11025	544
11	8000	492
12	7350	479
13	reserved	reserved
14	reserved	reserved
15	reserved	reserved

Table 7.8 — linebits

HILNmaxNumLine	0	1	2..3	4..7	8..15	16..31	32..63	64..127	128..255
linebits	0	1	2	3	4	5	6	7	8

Table 7.9 — HILNcontMode

HILNcontMode	additional decoder line continuation (see Subclause 7.5.1.4.3.1)
0	harmonic lines <-> individual lines and harmonic lines <-> harmonic lines
1	mode 0 plus individual lines <-> individual lines
2	no additional decoder line continuation
3	(reserved)

The number of frequency enhancement bits (fEnhbits[i]) in HILNenhaFrame() (see subclause 7.3.2.2) is calculated as follows:

- individual line, see INDIenhaPara():

```
fEnhbits[i] = max(0,fEnhbitsBase[ILFreqIndex[i]] + fEnhbitsMode[HILNenhaQuantMode])
```

- harmonic line, see HARMenhaPara():

```
fEnhbits[i] = max(0,fEnhbitsBase[harmFreqIndex] + fEnhbitsMode[HILNenhaQuantMode] +
                   fEnhbitsHarm[i])
```

Table 7.10 — fEnhbitsBase

ILFreqIndex	harmFreqIndex	fEnhbitsBase
0..159	0..1243	0
160..269	1244..1511	1
270..380	1512..1779	2
381..491	1780..2047	3
492..602		4
603..713		5
714..890		6

Table 7.11 — fEnhbitsMode

HILNenhaQuantMode	0	1	2	3
fEnhbitsMode	-3	-2	-1	0

Table 7.12 — fEnhbitsHarm

i	0	1	2..3	4..7	8..9
fEnhbitsHarm[i]	0	1	2	3	4

Table 7.13 — HILN constants

tmbits	4
atkbits	4
decbits	4
tmEnhbits	3
atkEnhbits	2
decEnhbits	2
phasebits	5

7.3.2 Bitstream frame (slPacketPayload)

The dynamic data for parametric coding is transmitted as SL packet payload in the base layer and the optional enhancement or extension layer Elementary Stream.

Parametric Base Layer -- Access Unit payload
For the parametric coder in unscalable mode or for the base layer in HILN scalable mode the following bitstream frame payload is defined:

```
slPacketPayload {
   PARAframe();
}
```

Parametric HILN Enhancement / Extension Layer -- Access Unit payload
To parse and decode the HILN enhancement layer, information decoded from the HILN base layer is required.

To parse and decode an HILN extension layer, information decoded from the HILN base layer and possible lower HILN extension layers is required. The bitstream syntax of the HILN extension layers is described in a way which requires the HILN base and extension bitstream frames being parsed in proper order:

1. HILNbasicFrame() base bitstream frame

2. HILNextFrame(1) 1st extension bitstream frame (if base bitstream frame available)

3. HILNextFrame(2) 2nd extension bitstream frame (if base and 1st extension bitstream frame available)

4. etc.

For the enhancement layer and the extension layer in HILN scalable mode the following bitstream frame payload is defined:

```
slPacketPayload {
    HILNenexFrame();
}
```

7.3.2.1 Parametric Audio bitstream frame

Table 7.14 — Syntax of PARAframe()

Syntax	No. of bits	Mnemonic

PARAframe()
{
if (PARAmode == 0) {
ErHVXCframe(HVXCrate);
}
else if (PARAmode == 1) {
HILNframe();
}
else if (PARAmode == 2) {
switchFrame();
}
else if (PARAmode == 3) {
mixFrame();
}
}		

Table 7.15 — Syntax of switchFrame()

Syntax	No. of bits	Mnemonic

switchFrame()
{
PARAswitchMode;
if (PARAswitchMode == 0) {
ErHVXCdoubleframe(HVXCrate);
}
else {
HILNframe();
}
}
	1	**uimsbf**

One of the following PARAswitchModes is selected in each frame:

Table 7.16 — PARAswitchMode

PARAswitchMode	Description
0	HVXC only
1	HILN only

Table 7.17 — Syntax of mixFrame()

Syntax	No. of bits	Mnemonic

mixFrame()
{
PARAmixMode;
if (PARAmixMode == 0) {
ErHVXCdoubleframe(HVXCrate);
}
else if (PARAmixMode == 1) {
HILNframe();
ErHVXCdoubleframe(2000);
}
else if (PARAmixMode == 2) {
HILNframe();
ErHVXCdoubleframe(4000);
}
else if (PARAmixMode == 3) {
HILNframe();
}
}
	2	**uimsbf**

One of the following PARAmixModes is selected in each frame:

Table 7.18 — PARAmixMode

PARAmixMode	Description
0	HVXC only
1	HVXC 2 kbit/s & HILN
2	HVXC 4 kbit/s & HILN
3	HILN only

Table 7.19 — Syntax of HVXCdoubleframe()

Syntax	No. of bits	Mnemonic

ErHVXCdoubleframe(rate)
{
if (rate >= 3000) {
ErHVXCfixframe(4000);
ErHVXCfixframe(rate);
}
else {
ErHVXCfixframe(2000);
ErHVXCfixframe(rate);
}
}		

7.3.2.2 HILN bitstream frame

Table 7.20 — Syntax of HILNframe()

Syntax	No. of bits	Mnemonic
HILNframe()		
{		
numLayer = 0;		
HILNbasicFrameESC0();		
HILNbasicFrameESC1();		
HILNbasicFrameESC2();		
HILNbasicFrameESC3();		
HILNbasicFrameESC4();		
layNumLine[0] = numLine;		
layPrevNumLine[0] = prevNumLine;		
for (k = 0; k < prevNumLine; k++) {		
layPrevLineContFlag[0][k] = prevLineContFlag[k];		
}		
}		

Table 7.21 — Syntax of HILNbasicFrameESC0()

Syntax	No. of bits	Mnemonic
HILNbasicFrameESC0()		
{		
prevNumLine = numLine;		
/* prevNumLine is set to the number of lines */		
/* in the previous frame */		
/* prevNumLine = 0 for the first bitstream frame */		
numLine;	**linebits**	**uimsbf**
harmFlag;	**1**	**uimsbf**
noiseFlag;	**1**	**uimsbf**
envFlag;	**1**	**uimsbf**
phaseFlag;	**1**	**uimsbf**
maxAmplIndexCoded;	**4**	**uimsbf**
maxAmplIndex = 4*maxAmplIndexCoded;		
if (harmFlag) {		
HARMbasicParaESC0();		
}		
if (noiseFlag) {		
NOISEbasicParaESC0();		
}		
}		

Table 7.22 — Syntax of HILNbasicFrameESC1()

Syntax	No. of bits	Mnemonic
HILNbasicFrameESC1()		
{		
if (harmFlag) {		
HARMbasicParaESC1();		
}		
if (noiseFlag) {		
NOISEbasicParaESC1();		
}		
INDIbasicParaESC1();		
}		

Table 7.23 — Syntax of HILNbasicFrameESC2()

Syntax	No. of bits	Mnemonic

HILNbasicFrameESC2()
{
INDIbasicParaESC2 ();
}		

Table 7.24 — Syntax of HILNbasicFrameESC3()

Syntax	No. of bits	Mnemonic

HILNbasicFrameESC3()
{
if (envFlag) {
envTmax;
envRatk;
envRdec;
}
if (harmFlag) {
HARMbasicParaESC3();
}
if (noiseFlag) {
NOISEbasicParaESC3();
}
INDIbasicParaESC3();
if (harmFlag) {
harmFreqStretch;
}
}
	
	tmbits
	atkbits
	decbits
	
	1..7
		
		uimsbf
		uimsbf
		uimsbf
		
		HFS

Table 7.25 — Syntax of HILNbasicFrameESC4()

Syntax	No. of bits	Mnemonic

HILNbasicFrameESC4()
{
if (harmFlag) {
HARMbasicParaESC4();
}
if (noiseFlag) {
NOISEbasicParaESC4();
}
INDIbasicParaESC4();
}		

Table 7.26 — Syntax of HARMbasicParaESC0()

Syntax	No. of bits	Mnemonic
HARMbasicParaESC0()		
{		
prevHarmAmplIndex = harmAmplIndex;		
prevHarmFreqIndex = harmFreqIndex;		
harmContFlag;	**1**	**uimsbf**
harmEnvFlag;	**1**	**uimsbf**
if (!harmContFlag) {		
harmAmplRel;	**6**	**uimsbf**
harmAmplIndex = maxAmplIndex + harmAmplRel;		
harmFreqIndex;	**11**	**uimsbf**
}		
}		

Table 7.27 — Syntax of HARMbasicParaESC1()

Syntax	No. of bits	Mnemonic
HARMbasicParaESC1()		
{		
numHarmParaIndex;	**4**	**uimsbf**
numHarmPara = numHarmParaTable[numHarmParaIndex];		
numHarmLineIndex;	**5**	**uimsbf**
numHarmLine = numHarmLineTable[numHarmLineIndex];		
if (harmContFlag) {		
contHarmAmpl	**3..8**	**DIA**
harmAmplIndex = prevHarmAmplIndex + contHarmAmpl;		
contHarmFreq	**2..9**	**DHF**
harmFreqIndex = prevHarmFreqIndex + contHarmFreq;		
}		
for (i = 0; i < 2; i++) {		
harmLAR[i];	**4..19**	**LARH1**
}		
}		

Table 7.28 — Syntax of HARMbasicParaESC3()

Syntax	No. of bits	Mnemonic
HARMbasicParaESC3()		
{		
for (i = 2; i < min(7,numHarmPara); i++) {		
harmLAR[i];	**3..18**	**LARH2**
}		
for (i = 7; i < numHarmPara; i++) {		
harmLAR[i];	**2..17**	**LARH3**
}		
}		

Table 7.29 — Syntax of HARMbasicParaESC4()

Syntax	No. of bits	Mnemonic
HARMbasicParaESC4()		
{		
if (phaseFlag && ! harmContFlag) {		
numHarmPhase;	**6**	**uimsbf**
}		
else {		
numHarmPhase = 0;		
}		
for (i = 0; i < numHarmPhase; i++) {		
harmPhase[i];	**phasebits**	**uimsbf**
harmPhaseAvail[i] = 1;		
}		
for (i = numHarmPhase; i < numHarmLine; i++) {		
harmPhaseAvail[i] = 0;		
}		
}		

Table 7.30 — numHarmParaTable

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
numHarmParaTable[i]	2	3	4	5	6	7	8	9	11	13	15	17	19	21	23	25

Table 7.31 — numHarmLineTable

i	0	1	2	3	4	5	6	7
numHarmLineTable[i]	3	4	5	6	7	8	9	10
i	8	9	10	11	12	13	14	15
numHarmLineTable[i]	12	14	16	19	22	25	29	33
i	16	17	18	19	20	21	22	23
numHarmLineTable[i]	38	43	49	56	64	73	83	94
i	24	25	26	27	28	29	30	31
numHarmLineTable[i]	107	121	137	155	175	197	222	250

Table 7.32 — Syntax of NOISEbasicParaESC0()

Syntax	No. of bits	Mnemonic
NOISEbasicParaESC0()		
{		
prevNoiseAmplIndex = noiseAmplIndex;		
noiseContFlag;	**1**	**uimsbf**
noiseEnvFlag;	**1**	**uimsbf**
if (!noiseContFlag) {		
noiseAmplRel;	**6**	**uimsbf**
noiseAmplIndex = maxAmplIndex + noiseAmplRel;		
}		
}		

Table 7.33 — Syntax of NOISEbasicParaESC1()

Syntax	No. of bits	Mnemonic
NOISEbasicParaESC1()		
{		
if (noiseContFlag) {		
contNoiseAmpl;	**3..8**	**DIA**
noiseAmplIndex = prevNoiseAmplIndex + contNoiseAmpl;		
}		
numNoiseParaIndex;	**4**	**uimsbf**
numNoisePara = numNoiseParaTable[numNoiseParaIndex];		
for (i = 0; i < min(2,numNoisePara); i++) {		
noiseLAR[i];	**2..17**	**LARN1**
}		
}		

Table 7.34 — Syntax of NOISEbasicParaESC3()

Syntax	No. of bits	Mnemonic
NOISEbasicParaESC3()		
{		
for (i = 2; i < numNoisePara; i++) {		
noiseLAR[i];	**1..18**	**LARN2**
}		
}		

Table 7.35 — Syntax of NOISEbasicParaESC4()

Syntax	No. of bits	Mnemonic
NOISEbasicParaESC4()		
{		
if (noiseEnvFlag) {		
noiseEnvTmax;	**tmbits**	**uimsbf**
noiseEnvRatk;	**atkbits**	**uimsbf**
noiseEnvRdec;	**decbits**	**uimsbf**
}		
}		

Table 7.36 — numNoiseParaTable

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
numNoiseParaTable[i]	1	2	3	4	5	6	7	9	11	13	15	17	19	21	23	25

Table 7.37 — Syntax of INDIbasicParaESC1()

Syntax	No. of bits	Mnemonic
INDIbasicParaESC1()		
{		
for (k = 0; k < prevNumLine; k++) {		
prevLineContFlag[k];	**1**	**uimsbf**
}		
i = 0;		
for (k = 0; k < prevNumLine; k++) {		
if (prevLineContFlag[k]) {		

```
                linePred[i] = k;
                lineContFlag[i++] = 1;
            }
        }
        while (i < numLine) {
            lineContFlag[i++] = 0;
        }
}
```

Table 7.38 — Syntax of INDIbasicParaESC2()

Syntax	No. of bits	Mnemonic
INDIbasicParaESC2()		
{		
lastNLFreq = 0;		
for (i = 0; i < prevNumLine; i++) {		
prevILFreqIndex[i] = ILFreqIndex[i];		
prevILAmplIndex[i] = ILAmplIndex[i];		
}		
for (i = 0; i < numLine; i++) {		
if (envFlag) {		
lineEnvFlag[i];	**1**	**uimsbf**
}		
}		
for (i = 0; i < numLine; i++) {		
if (!lineContFlag[i]) {		
if (numLine-1-i < 7) {		
ILFreqInc[i];	**0..14**	**SDC**
/* SDCdecode (maxFindex-lastNLFreq, */		
/* sdcILFTable[numLine-1-i]) */		
}		
else {		
ILFreqInc[i];	**0..14**	**SDC**
/* SDCdecode (maxFindex-lastNLFreq, */		
/* sdcILFTable[7]) */		
}		
ILFreqIndex[i] = lastNLFreq + ILFreqInc[i];		
lastNLFreq = ILFreqIndex[i];		
if (HILNquantMode) {		
ILAmplRel[i];	**4..10**	**SDC**
/* SDCdecode (50, sdcILATable) */		
ILAmplIndex[i] = maxAmplIndex + ILAmplRel[i];		
}		
else {		
ILAmplRel[i];	**3..9**	**SDC**
/* SDCdecode (25, sdcILATable) */		
ILAmplIndex[i] = maxAmplIndex + 2*ILAmplRel[i];		
}		
}		
}		
}		

Table 7.39 — Syntax of INDIbasicParaESC3()

Syntax	No. of bits	Mnemonic
INDIbasicParaESC3()		
{		
for (i = 0; i < numLine; i++) {		
if (lineContFlag[i]) {		
DILFreq[i];	**2..10**	**DIF**
ILFreqIndex[i] = prevILFreqIndex[linePred[i]] + DILFreq[i];		
DILAmpl[i];	**3..8**	**DIA**
ILAmplIndex[i] = prevILAmplIndex[linePred[i]] + DILAmpl[i];		
}		
}		
}		

Table 7.40 — Syntax of INDIbasicParaESC4()

Syntax	No. of bits	Mnemonic
INDIbasicParaESC4()		
{		
if (phaseFlag) {		
numLinePhase;	**linebits**	**uimsbf**
}		
else {		
numLinePhase = 0;		
}		
j = 0;		
for (i = 0; i < numLine; i++) {		
if (! linePred[i] && j<numLinePhase) {		
linePhase[i];	**phasebits**	**uimsbf**
linePhaseAvail[i] = 1;		
j++;		
}		
else {		
linePhaseAvail[i] = 0;		
}		
}		
}		

Table 7.41 — Syntax of HILNenexFrame()

Syntax	No. of bits	Mnemonic
HILNenexFrame()		
{		
/* HILNenhaLayer value in ParametricSpecificConfig() of */		
/* this Elementary Stream must be used here! */		
if (HILNenhaLayer) {		
HILNenhaFrame();		
}		
else {		
numLayer++;		
HILNextFrame(numLayer);		
}		
}		

Table 7.42 — Syntax of HILNenhaFrame()

Syntax	No. of bits	Mnemonic
HILNenhaFrame()		
{		
if (envFlag) {		
envTmaxEnha;	**tmEnhbits**	**uimsbf**
envRatkEnha;	**atkEnhbits**	**uimsbf**
envRdecEnha;	**decEnhbits**	**uimsbf**
}		
if (harmFlag) {		
HARMenhaPara();		
}		
INDIenhaPara();		
}		

Table 7.43 — Syntax of HARMenhaPara()

Syntax	No. of bits	Mnemonic
HARMenhaPara()		
{		
for (i = 0; i < min(numHarmLine,10); i++) {		
harmFreqEnha[i];	**fEnhbits[i]**	**uimsbf**
harmPhase[i];	**phasebits**	**uimsbf**
}		
}		

Table 7.44 — Syntax of INDIenhaPara()

Syntax	No. of bits	Mnemonic
INDIenhaPara()		
{		
for (i = 0; i < numLine; i++) {		
lineFreqEnha[i];	**fEnhbits[i]**	**uimsbf**
linePhase[i];	**phasebits**	**uimsbf**
}		
}		

Table 7.45 — Syntax of HILNextFrame()

Syntax	No. of bits	Mnemonic
HILNextFrame(numLayer)		
{		
layPrevNumLine[numLayer] = layNumLine[numLayer];		
/* layPrevNumLine[numLayer] = 0 for the */		
/* first bitstream frame */		
addNumLine[numLayer];	**linebits**	**uimsbf**
if (phaseFlag) {		
layNumLinePhase[numLayer];	**linebits**	**uimsbf**
}		
layNumLine[numLayer] = layNumLine[numLayer-1] + addNumLine[numLayer];		
for (k = 0; k < layPrevNumLine[numLayer-1]; k++) {		
if (layPrevLineContFlag[numLayer-1][k]) {		
layPrevLineContFlag[numLayer][k] = 1;		
}		

```
            else {
                layPrevLineContFlag[numLayer][k];          1          uimsbf
            }
        }
        for (k = layPrevNumLine[numLayer-1];
                k < layPrevNumLine[numLayer]; k++) {
            layPrevLineContFlag[numLayer][k];              1          uimsbf
        }
        i = layNumLine[numLayer-1];
        for (k = 0; k < layPrevNumLine[numLayer-1]; k++) {
            if (!layPrevLineContFlag[numLayer-1][k] &&
                    layPrevLineContFlag[numLayer][k]) {
                linePred[i] = k;
                lineContFlag[i++] = 1
            }
        }
        for (k = layPrevNumLine[numLayer-1];
                k < layPrevNumLine[numLayer]; k++) {
            if (layPrevLineContFlag[numLayer][k]) {
                linePred[i] = k;
                lineContFlag[i++] = 1;
            }
        }
        while (i < layNumLine[numLayer]) {
            lineContFlag[i++] = 0;
        }
        INDIextPara(numLayer);
        if (phaseFlag) {
            INDIextPhasePara(numLayer);
        }
}
```

Table 7.46 — Syntax of INDIextPara()

Syntax	No. of bits	Mnemonic
INDIextPara(numLayer)		
{		
lastNLFreq = 0;		
for (i = layPrevNumLine[numLayer-1];		
i < layPrevNumLine[numLayer]; i++) {		
prevILFreqIndex[i] = ILFreqIndex[i];		
prevILAmplIndex[i] = ILAmplIndex[i];		
}		
for (i = layNumLine[numLayer-1];		
i < layNumLine[numLayer]; i++) {		
if (envFlag) {		
lineEnvFlag[i];	**1**	**uimsbf**
}		
if (lineContFlag[i]) {		
DILFreq[i];	**2..10**	**DIF**
ILFreqIndex[i] = prevILFreqIndex[linePred[i]] + DILFreq[i];		
DILAmpl[i];	**3..8**	**DIA**
ILAmplIndex[i] = prevILAmplIndex[linePred[i]] + DILAmpl[i];		
}		
else {		
if (layNumLine[numLayer]-1-i < 7) {		

```
                    ILFreqInc[i];                                   0..14       SDC
                    /* SDCdecode (maxFindex-lastNLFreq,         */
                    /* sdcILFTable[layNumLine[numLayer]-1-i]) */
                }
                else {
                    ILFreqInc[i];                                   0..14       SDC
                    /* SDCdecode (maxFindex-lastNLFreq, */
                    /* sdcILFTable[7]) */
                }
                ILFreqIndex[i] = lastNLFreq + ILFreqInc[i];
                lastNLFreq = ILFreqIndex[i];
                if (HILNquantMode) {
                    ILAmplRel[i];                                   4..10       SDC
                    /* SDCdecode (50, sdcILATable) */
                    ILAmplIndex[i] = maxAmplIndex + ILAmplRel[i];
                }
                else {
                    ILAmplRel[i];                                   3..9        SDC
                    /* SDCdecode (25, sdcILATable) */
                    ILAmplIndex[i] = maxAmplIndex +
                                        2*ILAmplRel[i];
                }
            }
        }
}
```

Table 7.47 — Syntax of INDIextPhasePara()

Syntax	No. of bits	Mnemonic

INDIextPhasePara(numLayer)
{
j = 0;
for (i = layNumLine[numLayer-1]; i < layNumLine[numLayer];
i++) {
if (! linePred[i] && j < layNumLinePhase[numLayer]) {
linePhase[i];
linePhaseAvail[i] = 1;
j++;
}
else {
linePhaseAvail[i] = 0;
}
}
}
	phasebits	**uimsbf**

7.3.2.3 HILN codebooks

Table 7.48 — LARH1 code (harmLAR[0..1])

codeword	harmLAR[i]	codeword	harmLAR[i]
1000000000000000100	-6.350	0100	0.050
1000000000000000101	-6.250	0101	0.150
1000000000000000110	-6.150	0110	0.250
1000000000000000111	-6.050	0111	0.350
100000000000000100	-5.950	00100	0.450
100000000000000101	-5.850	00101	0.550

1000000000000000110	-5.750	00110	0.650
1000000000000000111	-5.650	00111	0.750
10000000000000100	-5.550	000100	0.850
10000000000000101	-5.450	000101	0.950
10000000000000110	-5.350	000110	1.050
10000000000000111	-5.250	000111	1.150
1000000000000100	-5.150	0000100	1.250
1000000000000101	-5.050	0000101	1.350
1000000000000110	-4.950	0000110	1.450
1000000000000111	-4.850	0000111	1.550
100000000000100	-4.750	00000100	1.650
100000000000101	-4.650	00000101	1.750
100000000000110	-4.550	00000110	1.850
100000000000111	-4.450	00000111	1.950
10000000000100	-4.350	000000100	2.050
10000000000101	-4.250	000000101	2.150
10000000000110	-4.150	000000110	2.250
10000000000111	-4.050	000000111	2.350
1000000000100	-3.950	0000000100	2.450
1000000000101	-3.850	0000000101	2.550
1000000000110	-3.750	0000000110	2.650
1000000000111	-3.650	0000000111	2.750
100000000100	-3.550	00000000100	2.850
100000000101	-3.450	00000000101	2.950
100000000110	-3.350	00000000110	3.050
100000000111	-3.250	00000000111	3.150
10000000100	-3.150	000000000100	3.250
10000000101	-3.050	000000000101	3.350
10000000110	-2.950	000000000110	3.450
10000000111	-2.850	000000000111	3.550
1000000100	-2.750	0000000000100	3.650
1000000101	-2.650	0000000000101	3.750
1000000110	-2.550	0000000000110	3.850
1000000111	-2.450	0000000000111	3.950
100000100	-2.350	00000000000100	4.050
100000101	-2.250	00000000000101	4.150
100000110	-2.150	00000000000110	4.250
100000111	-2.050	00000000000111	4.350
10000100	-1.950	000000000000100	4.450
10000101	-1.850	000000000000101	4.550
10000110	-1.750	000000000000110	4.650
10000111	-1.650	000000000000111	4.750
1000100	-1.550	0000000000000100	4.850
1000101	-1.450	0000000000000101	4.950
1000110	-1.350	0000000000000110	5.050
1000111	-1.250	0000000000000111	5.150
100100	-1.150	00000000000000100	5.250
100101	-1.050	00000000000000101	5.350
100110	-0.950	00000000000000110	5.450
100111	-0.850	00000000000000111	5.550
10100	-0.750	000000000000000100	5.650
10101	-0.650	000000000000000101	5.750
10110	-0.550	000000000000000110	5.850
10111	-0.450	000000000000000111	5.950
1100	-0.350	0000000000000000100	6.050
1101	-0.250	0000000000000000101	6.150

1110	-0.150	0000000000000000110	6.250
1111	-0.050	0000000000000000111	6.350

Table 7.49 — LARH2 code (harmLAR[2..6])

codeword	harmLAR[.]	codeword	harmLAR[.]
100000000000000010	-4.725	010	0.075
100000000000000011	-4.575	011	0.225
10000000000000010	-4.425	0010	0.375
10000000000000011	-4.275	0011	0.525
1000000000000010	-4.125	00010	0.675
1000000000000011	-3.975	00011	0.825
100000000000010	-3.825	000010	0.975
100000000000011	-3.675	000011	1.125
10000000000010	-3.525	0000010	1.275
10000000000011	-3.375	0000011	1.425
1000000000010	-3.225	00000010	1.575
1000000000011	-3.075	00000011	1.725
100000000010	-2.925	000000010	1.875
100000000011	-2.775	000000011	2.025
10000000010	-2.625	0000000010	2.175
10000000011	-2.475	0000000011	2.325
1000000010	-2.325	00000000010	2.475
1000000011	-2.175	00000000011	2.625
100000010	-2.025	000000000010	2.775
100000011	-1.875	000000000011	2.925
10000010	-1.725	0000000000010	3.075
10000011	-1.575	0000000000011	3.225
1000010	-1.425	00000000000010	3.375
1000011	-1.275	00000000000011	3.525
100010	-1.125	000000000000010	3.675
100011	-0.975	000000000000011	3.825
10010	-0.825	0000000000000010	3.975
10011	-0.675	0000000000000011	4.125
1010	-0.525	00000000000000010	4.275
1011	-0.375	00000000000000011	4.425
110	-0.225	000000000000000010	4.575
111	-0.075	000000000000000011	4.725

Table 7.50 — LARH3 code (harmLAR[7..24])

codeword	harmLAR[.]	codeword	harmLAR[.]
10000000000000001	-2.325	01	0.075
1000000000000001	-2.175	001	0.225
100000000000001	-2.025	0001	0.375
10000000000001	-1.875	00001	0.525
1000000000001	-1.725	000001	0.675
100000000001	-1.575	0000001	0.825
10000000001	-1.425	00000001	0.975
1000000001	-1.275	000000001	1.125
100000001	-1.125	0000000001	1.275
10000001	-0.975	00000000001	1.425
1000001	-0.825	000000000001	1.575
100001	-0.675	0000000000001	1.725

10001	-0.525	00000000000001	1.875
1001	-0.375	000000000000001	2.025
101	-0.225	0000000000000001	2.175
11	-0.075	00000000000000001	2.325

Table 7.51 — LARN1 code (noiseLAR[0,1])

codeword	noiseLAR[.]	codeword	noiseLAR[.]
10000000000000001	-4.65	01	0.15
1000000000000001	-4.35	001	0.45
100000000000001	-4.05	0001	0.75
10000000000001	-3.75	00001	1.05
1000000000001	-3.45	000001	1.35
100000000001	-3.15	0000001	1.65
10000000001	-2.85	00000001	1.95
1000000001	-2.55	000000001	2.25
100000001	-2.25	0000000001	2.55
10000001	-1.95	00000000001	2.85
1000001	-1.65	000000000001	3.15
100001	-1.35	0000000000001	3.45
10001	-1.05	00000000000001	3.75
1001	-0.75	000000000000001	4.05
101	-0.45	0000000000000001	4.35
11	-0.15	00000000000000001	4.65

Table 7.52 — LARN2 code (noiseLAR[2..24])

codeword	noiseLAR[.]	codeword	noiseLAR[.]
110000000000000001	-6.35	101	0.35
11000000000000001	-5.95	1001	0.75
1100000000000001	-5.55	10001	1.15
110000000000001	-5.15	100001	1.55
11000000000001	-4.75	1000001	1.95
1100000000001	-4.35	10000001	2.35
110000000001	-3.95	100000001	2.75
11000000001	-3.55	1000000001	3.15
1100000001	-3.15	10000000001	3.55
110000001	-2.75	100000000001	3.95
11000001	-2.35	1000000000001	4.35
1100001	-1.95	10000000000001	4.75
110001	-1.55	100000000000001	5.15
11001	-1.15	1000000000000001	5.55
1101	-0.75	10000000000000001	5.95
111	-0.35	100000000000000001	6.35
0	0.00		

Table 7.53 — DIA code

codeword	value	codeword	value
111 1 1111	-25	001	1
111 1 1110	-24	011 0	2
111 1 1101	-23	100 0	3
111 1 xxxx	-y	101 0 0	4
111 1 0001	-11	101 0 1	5
111 1 0000	-10	110 0 00	6
110 1 11	-9	110 0 01	7
110 1 10	-8	110 0 10	8
110 1 01	-7	110 0 11	9
110 1 00	-6	111 0 0000	10
101 1 1	-5	111 0 0001	11
101 1 0	-4	111 0 xxxx	y
100 1	-3	111 0 1101	23
011 1	-2	111 0 1110	24
010	-1	111 0 1111	25
000	0		

Table 7.54 — DIF code

codeword	value	codeword	value
11 11 1 11111	-42	01 0	1
11 11 1 11110	-41	10 0 0	2
11 11 1 11101	-40	10 0 1	3
11 11 1 xxxxx	-y	11 00 0	4
11 11 1 00001	-12	11 01 0 0	5
11 11 1 00000	-11	11 01 0 1	6
11 10 1 11	-10	11 10 0 00	7
11 10 1 10	-9	11 10 0 01	8
11 10 1 01	-8	11 10 0 10	9
11 10 1 00	-7	11 10 0 11	10
11 01 1 1	-6	11 11 0 00000	11
11 01 1 0	-5	11 11 0 00001	12
11 00 1	-4	11 11 0 xxxxx	y
10 1 1	-3	11 11 0 11101	40
10 1 0	-2	11 11 0 11110	41
01 1	-1	11 11 0 11111	42
00	0		

Table 7.55 — DHF code

codeword	value	codeword	value
11 1 111111	-69	01 0	1
11 1 111110	-68	10 0 00	2
11 1 111101	-67	10 0 01	3
11 1 xxxxxx	-y	10 0 10	4
11 1 000001	-7	10 0 11	5
11 1 000000	-6	11 0 000000	6
10 1 11	-5	11 0 000001	7
10 1 10	-4	11 0 xxxxxx	y
10 1 01	-3	11 0 111101	67
10 1 00	-2	11 0 111110	68
01 1	-1	11 0 111111	69
00	0		

Table 7.56 — HFS code

codeword	value	codeword	value
1 1 1 1111	-17	1 0 0	1
1 1 1 1110	-16	1 0 1 0000	2
1 1 1 1101	-15	1 0 1 0001	3
1 1 1 xxxx	-y	1 0 1 xxxx	y
1 1 1 0001	-3	1 0 1 1101	15
1 1 1 0000	-2	1 0 1 1110	16
1 1 0	-1	1 0 1 1111	17
0	0		

Notes on Table 7.53 to Table 7.56: The grouping of bits within a codeword (e.g. “1 1 1 1111”) is provided for easier readability only. Codewords not explicitly listed in the codebooks (e.g. “1 1 1 xxxx”) are defined by incrementing the implicit part of the codeword “xxxx” (uimsbf) and the magnitude “y” of the corresponding value. In all cases, the codewords and values for the two smallest and the three largest magnitudes are listed explicitly.

7.3.2.4 HILN SubDivisionCode (SDC)

The SubDivisionCode (SDC) is an algorithmically generated variable length code, based on a given table and a given number of different codewords. The decoding process is defined below.

The idea behind this coding scheme is the subdivision of the probability density function into two parts which represent an equal probability. One bit is transmitted that determines the part the value to be coded is located. This subdivision is repeated until the width of the part is one and then its position is equal to the value being coded. The positions of the boundaries are taken out of a table of 32 quantized, fixed point values. Besides this table (parameter “tab”) the number of different codewords (parameter “k”) is needed too.

The following C function SDCDecode(k, tab) together with the 9 tables sdcILATable[32] and sdcILFTable[8][32] describe the decoding. The function GetBit() returns the next bit in the stream.

```
int sdcILATable[32] = {
    0,  13,  27,  41,  54,  68,  82,  96, 110, 124, 138, 152, 166, 180, 195, 210,
  225, 240, 255, 271, 288, 305, 323, 342, 361, 383, 406, 431, 460, 494, 538, 602
};

int sdcILFTable[8][32] = {
{   0,  53,  87, 118, 150, 181, 212, 243, 275, 306, 337, 368, 399, 431, 462, 493,
  524, 555, 587, 618, 649, 680, 711, 743, 774, 805, 836, 867, 899, 930, 961, 992 },
{   0,  34,  53,  71,  89, 106, 123, 141, 159, 177, 195, 214, 234, 254, 274, 296,
  317, 340, 363, 387, 412, 438, 465, 494, 524, 556, 591, 629, 670, 718, 774, 847 },
{   0,  26,  41,  54,  66,  78,  91, 103, 116, 128, 142, 155, 169, 184, 199, 214,
  231, 247, 265, 284, 303, 324, 346, 369, 394, 422, 452, 485, 524, 570, 627, 709 },
{   0,  23,  35,  45,  55,  65,  75,  85,  96, 106, 117, 128, 139, 151, 164, 177,
  190, 204, 219, 235, 252, 270, 290, 311, 334, 360, 389, 422, 461, 508, 571, 665 },
{   0,  20,  30,  39,  48,  56,  64,  73,  81,  90,  99, 108, 118, 127, 138, 149,
  160, 172, 185, 198, 213, 228, 245, 263, 284, 306, 332, 362, 398, 444, 507, 608 },
{   0,  18,  27,  35,  43,  50,  57,  65,  72,  79,  87,  95, 104, 112, 121, 131,
  141, 151, 162, 174, 187, 201, 216, 233, 251, 272, 296, 324, 357, 401, 460, 558 },
{   0,  16,  24,  31,  38,  45,  51,  57,  64,  70,  77,  84,  91,  99, 107, 115,
  123, 132, 142, 152, 163, 175, 188, 203, 219, 237, 257, 282, 311, 349, 403, 493 },
{   0,  12,  19,  25,  30,  35,  41,  46,  51,  56,  62,  67,  73,  79,  85,  92,
   99, 106, 114, 122, 132, 142, 153, 165, 179, 195, 213, 236, 264, 301, 355, 452 }
};
```

```
int SDCDecode (int k, int *tab)
{
    int *pp;
    int g,dp,min,max;

    min = 0;
    max = k - 1;
    pp = tab + 16;
    dp = 16;

    while (min != max)
    {
        if (dp) g = (k*(*pp)) >> 10; else g =(max+min) >> 1;
        dp >>= 1;
        if (GetBit() == 0) { pp -= dp; max = g; } else { pp += dp; min = g + 1; }
    }
    return max;
}
```

7.4 Bitstream semantics

7.4.1 Decoder configuration (ParametricSpecificConfig)

Bitstream elements:

isBaseLayer A one-bit identifier representing whether the corresponding layer is the base layer (1) or an enhancement or extension layer (0).

7.4.1.1 Parametric Audio decoder configuration

Bitstream elements:

PARAmode A 2 bit field indicating parametric coder operation mode.

PARAextensionFlag A flag indicating the presence of MPEG-4 version 3 data (for future use).

7.4.1.2 HILN decoder configuration

Bitstream elements:

HILNquantMode A 1 bit field indicating the individual line quantizer mode.

HILNmaxNumLine An 8 bit field indicating the maximum number of individual lines in a bitstream frame. It also determines linebits field size (see Table 7.8).

HILNsampleRateCode A 4 bit field indicating the sampling rate used for HILN parameter decoding (see Table 7.7).

HILNframeLength A 12 bit field indicating the HILN frame length in samples at the sampling rate indicated by HILNsampleRateCode.

HILNcontMode A 2 bit field indicating the additional decoder line continuation mode (see Table 7.9).

HILNenhaLayer A flag indicating whether this Elementary Stream contains an enhancement layer or an extension layer.

HILNenhaQuantMode A 2 bit field indicating frequency enhancement quantizer mode (see Table 7.11).

7.4.2 Bitstream frame (slPacketPayload)

7.4.2.1 Parametric Audio bitstream frame

Bitstream elements:

PARAswitchMode A flag indicating which coding tool is used in the current frame of a HVXC/HILN switching bitstream (see Table 7.16).

PARAmixMode A 2 bit field indicating which coding tools are used in the current frame of a HVXC/HILN mixing bitstream (see Table 7.18).

7.4.2.2 HILN bitstream frame

Bitstream elements:

numLine A field indicating the number of individual lines in the current frame.

harmFlag A flag indicating the presence of harmonic line data in the current frame.

noiseFlag A flag indicating the presence of noise component data in the current frame.

phaseFlag A flag indicating the presence of line start phase data in the current frame.

numLinePhase A field indicating the number of individual lines with start phase in the current frame.

maxAmpIIndexCoded A field indicating the maximum amplitude of a new signal component in the current frame.

envFlag A flag indicating the presence of envelope data in the current frame.

envTmax Coded envelope parameter: time of maximum.

envRatk Coded envelope parameter: attack rate.

envRdec Coded envelope parameter: decay rate.

prevLineContFlag[k] A flag indicating that the k-th individual line of the previous frame is continued in the current frame.

numHarmParaIndex A field indicating the number of harmonic line LPC parameters in the current frame (see Table 7.30).

numHarmLineIndex A field indicating the number of harmonic lines in the current frame (see Table 7.31).

harmContFlag A flag indicating that the harmonic lines are continued from the previous frame.

numHarmPhase A field indicating the number of harmonic lines with start phase in the current frame.

harmEnvFlag A flag indicating that the amplitude envelope is applied to the harmonic lines.

contHarmAmpl Coded amplitude change of the harmonic lines (see Table 7.53).

contHarmFreq Coded fundamental frequency change of the harmonic lines (see Table 7.55).

harmAmpIRel Coded relative amplitude of the harmonic lines.

harmFreqIndex	Coded fundamental frequency of the harmonic lines.
harmFreqStretch	Coded frequency stretching parameter of the harmonic lines (see Table 7.56).
harmLAR[i]	Coded LAR LPC parameters of the harmonic lines (see Table 7.48, Table 7.49, Table 7.50).
harmPhase[i]	Coded phase of i-th harmonic line.
numNoiseParaIndex	A field indicating the number of noise LPC parameters in the current frame (see Table 7.36).
noiseContFlag	A flag indicating that the noise is continued from the previous frame.
noiseEnvFlag	A flag indicating that noise envelope data is present in the current frame.
contNoiseAmpl	Coded amplitude change of the noise (see Table 7.53).
noiseAmplRel	Coded relative amplitude of the noise.
noiseEnvTmax	Coded noise envelope parameter: time of maximum
noiseEnvRatk	Coded noise envelope parameter: attack rate.
noiseEnvRdec	Coded noise envelope parameter: decay rate.
noiseLAR[i]	Coded LAR LPC parameters of the noise (see Table 7.51, Table 7.52).
lineEnvFlag[i]	A flag indicating that the amplitude envelope is applied to the i-th individual line.
DILFreq[i]	Coded frequency change of i-th individual line (see Table 7.54).
DILAmpl[i]	Coded amplitude change of i-th individual line (see Table 7.53).
ILFreqInc[i]	Coded frequency increment of i-th individual line (see Subclause 7.3.2.4).
ILAmplRel[i]	Coded relative amplitude of i-th individual line (see Subclause 7.3.2.4).
linePhase[i]	Coded phase of i-th individual line.
envTmaxEnha	Coded envelope enhancement parameter: time of maximum.
envRatkEnha	Coded envelope enhancement parameter: attack rate.
envRdecEnha	Coded envelope enhancement parameter: decay rate.
harmFreqEnha[i]	Coded frequency enhancement of i-th harmonic line.
lineFreqEnha[i]	Coded frequency enhancement of i-th individual line.
addNumLine[i]	A field indicating the number of individual lines in extension layer i of the current frame.
layNumLinePhase[i]	A field indicating the number of individual lines with start phase in extension layer i of the current frame.
layPrevLineContFlag[i][k]	A flag indicating that the k-th individual line of the previous frame is continued in extension layer i of the current frame.

Help elements:

numLayer	The number of extension layers available (0 if only base layer available).
layNumLine[i]	The total number of individual lines in the current frame as conveyed in the base layer and the first i extension layers.
prevNumLine	The number of individual lines in the previous frame.
layPrevNumLine[i]	The total number of individual lines in the previous frame as conveyed in the base layer and the first i extension layers.
maxAmplIndex	The maximum amplitude of a new signal component in the current frame.
linePred[i]	Index of the predecessor in the previous frame of the i-th individual line in the current frame.
lineContFlag[i]	A flag indicating that line i in the current frame is continued from the previous frame.
numHarmPara	The number of harmonic line LPC parameters in the current frame.
numHarmLine	The number of harmonic lines in the current frame.
harmAmplIndex	Amplitude index of the harmonic lines in the current frame.
harmFreqIndex	Fundamental frequency index of the harmonic lines in the current frame.
prevHarmAmplIndex	Amplitude index of the harmonic lines in the previous frame.
prevHarmFreqIndex	Fundamental frequency index of the harmonic lines in the previous frame.
harmPhaseAvail[i]	A flag indicating that start phase information is available for the i-th harmonic line.
numNoisePara	The number of noise LPC parameters in the current frame.
noiseAmplIndex	Amplitude index of the noise in the current frame.
prevNoiseAmplIndex	Amplitude index of the noise in the previous frame.
lastNLFreq	Individual line frequency increment accumulator.
ILFreqIndex[i]	Frequency index of the i-th individual line in the current frame.
ILAmplIndex[i]	Amplitude index of the i-th individual line in the current frame.
prevILFreqIndex[i]	Frequency index of the i-th individual line in the previous frame.
prevILAmplIndex[i]	Amplitude index of the i-th individual line in the previous frame.
linePhaseAvail[i]	A flag indicating that start phase information is available for the i-th individual line.
linebits	Number of bits for numLine.
tmbits	Number of bits for envTmax.
atkbits	Number of bits for encRatk.
decbits	Number of bits for envRdec.
tmEnhbits	Number of bits for envTmaxEnha.

atkEnhbits Number of bits for encRatkEnha.

decEnhbits Number of bits for envRdecEnha.

fEnhbits[i] Number of bits for lineFreqEnha[i] and harmFreqEnha[i] (see Subclause 7.3.1.2).

phasebits Number of bits for linePhase and harmPhase.

7.5 Parametric decoder tools

7.5.1 HILN decoder tools

The Harmonic and Individual Lines plus Noise (HILN) decoder utilizes a set of parameters which are encoded in the bitstream to describe the audio signal. Three different signal models are supported:

Table 7.57 — HILN signal models

signal model	description	essential parameters
harmonic lines	group of sinusoidal signals with common fundamental frequency	fundamental frequency and amplitudes of the spectral lines
individual lines	sinusoidal signals	frequency and amplitude of the individual spectral lines
noise	spectrally shaped noise signal	spectral shape and power of the noise

The HILN decoder first reconstructs these parameters from the bitstream with a set of decoding tools and then synthesizes the audio signal based on these parameters using a set of synthesizer tools:

- harmonic line decoder
- individual line decoder
- noise decoder
- harmonic and individual line synthesizer
- noise synthesizer

The HILN decoder tools reconstruct the parameters of the harmonic and individual lines (frequency, amplitude) and the noise (spectral shape) as well as possible envelope parameters from the bitstream.

The HILN synthesizer tools reconstruct one frame of the audio signal based on the parameters decoded by the HILN decoder tools for the current bitstream frame.

The samples of the decoded audio signal have a full scale range of [-32768, 32767] and eventual outliers must be limited (“clipped”) to these values.

The HILN decoder supports a wide range of frame lengths and sampling frequencies. By scaling the synthesizer frame length with an arbitrary factor, speed change functionality is available at the decoder. By scaling the line frequencies and resampling the noise signal with an arbitrary factor, pitch change functionality is available at the decoder.

The HILN decoder can operate in two different modes, as basic decoder and as enhanced decoder. The basic decoder which is used for normal operation only evaluates the information available in the bitstream elements HILNbasicFrame() to reconstruct the audio signal. To allow large step scalability in combination with other coder tools (e.g. GA scalable) the additional bitstream elements HILNenhaFrame() need to be transmitted and the HILN decoder must operate in the enhanced mode which exploits the information of both HILNbasicFrame() and

HILNenhaFrame(). This mode reconstructs an audio signal with well defined phase relationships which can be combined with a residual signal coded at higher bitrates using an enhancement coder (e.g. GA scalable). If the HILN decoder is used in this way as a core for a scalable coder no noise signal must be synthesized for the signal which is given to the enhancement decoder.

Due to the parametric signal representation utilized by the HILN parametric coder, it is well suited for applications requiring bitrate scalable coding. HILN bitrate scalable coding is accomplished by supplementing the data encoded in an HILNbasicFrame() of the basic bitstream by data encoded in one or more HILNextFrame() of one or more extension bitstreams transmitted as additional Elementary Streams. It should be noted that the coding efficiency of a combined bitstream consisting of a basic and one or more extension bitstreams is slightly lower than the coding efficiency of an non-scalable basic bitstream having the same total bitrate.

7.5.1.1 Harmonic line decoder

7.5.1.1.1 Tool description

This tool decodes the parameters of the harmonic lines transmitted in the bitstream.

7.5.1.1.2 Definitions

prevNumHarmPara	The number of harmonic line LPC parameters in the previous frame.
harmLPCPara[i]	Harmonic line LPC parameter i in the current frame (LARs for harmonic tone spectrum).
prevHarmLPCPara[i]	Harmonic line LPC parameter i in the previous frame (LARs for harmonic tone spectrum).
hFreq	Fundamental frequency of the harmonic lines.
hStretch	Frequency stretching of the harmonic lines.
harmAmpl	Harmonic tone amplitude.
harmPwr	Harmonic tone power.
hLineAmpl[i]	Amplitude of i-th harmonic line.
hLineFreq[i]	Frequency of i-th harmonic line (in Hz).
hLineAmplEnh[i]	Enhanced amplitude of i-th harmonic line.
hLineFreqEnh[i]	Enhanced frequency of i-th harmonic line (in Hz).
hLinePhaseEnh[i]	Phase of i-th harmonic line (in rad).
ha[i]	Unscaled amplitude of i-th harmonic line.
r[i]	LPC reflection coefficients.
h[i]	LPC impulse response.
H(z)	LPC system function.

7.5.1.1.3 Decoding process

If the harmFlag is set and thus HARMbasicPara() data and in enhancement mode HARMenhaPara() data is available in the current frame, the parameters of the harmonic lines are decoded and dequantized as follows:

7.5.1.1.3.1 Basic decoder

A harmonic tone is represented by its fundamental frequency, its power and a set of LPC-Parameters.

First the harmNumPara LAR parameters are reconstructed. Prediction from the previous frame is used when harmContFlag is set.

```
float harmLPCMean[25] = { 5.0, -1.5, 0.0, 0.0, 0.0, ... , 0.0 };
float harmPredCoeff[25] = { 0.75, 0.75, 0.5, 0.5, 0.5, ... , 0.5 };

for (i = 0; i < numHarmPara; i++) {
   if (i < prevNumHarmPara && harmContFlag )
      pred = harmLPCMean[i] +
         (prevHarmLPCPara[i]-harmLPCMean[i])*harmPredCoeff[i];
   else
      pred = harmLPCMean[i];
   harmLPCPara[i] = pred + harmLAR[i];
}
```

Parameters needed in the following frame are stored in the frame-to-frame memory:

```
prevNumHarmPara = numHarmPara;
for (i = 0; i < numHarmPara; i++)
   prevHarmLPCPara[i] = harmLPCPara[i];
```

The fundamental frequency and stretching of the harmonic lines are dequantized:

```
hFreq = 20 * exp(log(4000./20.) * (harmFreqIndex+0.5) / 2048.0);
hStretch = harmFreqStretch / 16000.0;
```

The amplitude and power of the harmonic tone is dequantized as follows:

```
harmAmpl = 32768 * pow(10, -1.5*(harmAmplIndex+0.5)/20);
harmPwr = harmAmpl*harmAmpl;
```

The harmEnvFlag and harmContFlag flags require no further dequantization; they are directly passed on to the synthesizer tool.

The LPC-Parameters are transmitted in the form of “Logarithmic Area Ratios” (LAR) as described above. After decoding the parameters the frequencies and amplitudes of the harmNumLine partials of the harmonic tone are calculated as follows:

The frequencies of the harmonic lines are calculated:

```
for (i = 0; i < numHarmLine; i++)
   hLineFreq[i] = hFreq * (i+1) * (1 + hStretch*(i+1));
```

The LPC-Parameters represent an IIR-Filter. The amplitudes of the sinusoids are obtained by calculating the absolute value of this filter's system function H(z) at the corresponding frequencies.

```
for (i = 0; i < numHarmLine; i++)
   ha[i] = abs( H( exp( j * pi * (i+1)/(numHarmLine+1) ) );
```

with j = sqrt(-1) and

```
   H(z) = 1 / ( 1 - h[0]*pow(z,-1) - h[1]*pow(z,-2) - ... - h[numHarmPara-1]*pow(z,-
numHarmPara) )
```

The impulse response h[i] is calculated from the LARs by the following algorithm:

In a first step the LARs are converted to reflection coefficients:

```
for (i = 0; i < numHarmPara; i++)
   r[i] = ( exp(harmLPCPara[i]) - 1 ) / ( exp(harmLPCPara[i]) + 1 );
```

After this the reflection coefficients are converted to the time response. The C function given below does this conversion in place (call with x[i]=r[i] and N=numHarmPara; returns with x[i]=h[i]):

```
void Convert_k_to_h (float *x, int N)
{
   int i,j;
   float  a,b,c;

   for (i = 1; i < N; i++)
   {
      c = x[i];

      for (j = 0; j < i-j-1; j++)
      {
         a = x[ j ];
         b = x[i-j-1];
         x[j] = a-c*b;
         x[i-j-1] = b-c*a;
      }
      if (j == i-j-1)
         x[j] -= c*x[j];
   }
}
```

After calculating the amplitudes ha[i] they must be normalized and multiplied with harmAmpl to find the harmonic line amplitudes meeting the condition:

```
sum (hLineAmpl[i]*hLineAmpl[i]) = power of harmonic tone
```

This is realized as follows:

```
p = 0.0;
for (i = 0; i < numHarmLine; i++)
   p += ha[i]*ha[i];
s = sqrt( harmPwr / p );
for (i = 0; i < numHarmLine; i++)
   hLineAmpl[i] = ha[i] * s;
```

The optional phase information is decoded as follows:

```
for (i = 0; i < numHarmLine; i++) {
   if (harmPhaseAvail[i]) {
      hStartPhase[i] = 2*pi*(harmPhase[i]+0.5)/(1<<phasebits)-pi;
      hStartPhaseAvail[i] = 1;
   }
   else
      hStartPhaseAvail[i] = 0;
}
```

7.5.1.1.3.2 Enhanced decoder

In this mode, the harmonic line parameters decoded by the basic decoder are refined and also line phases are decoded using the information contained in HARMenhaPara() as follows:

For the first maximum 10 harmonic lines i

```
i = 0 .. min(numHarmLine,10)-1
```

the enhanced harmonic line parameters are calculated using the basic harmonic line parameters and the data in the enhancement bitstream:

```
hLineAmplEnh[i] = hLineAmpl[i];
hLineFreqEnh[i] = hLineFreq[i] *
       (1+((harmFreqEnh[i]+0.5)/(1<<fEnhbits[i])-0.5)*(hFreqRelStep-1));
```

where hFreqRelStep is the ratio of two neighboring fundamental frequency quantizer steps:

```
hFreqRelStep = exp(log(4000/20)/2048));
```

For both line types the phase is decoded from the enhancement bitstream:

```
hLinePhaseEnh[i] = 2*pi*(harmPhase[i]+0.5)/(1<<phasebits)-pi
```

7.5.1.2 Individual line decoder

7.5.1.2.1 Tool description

The individual line basic bitstream decoder reconstructs the line parameters frequency, amplitude, and envelope from the bitstream. The enhanced bitstream decoder reconstructs the line parameters frequency, amplitude, and envelope with finer quantization and additionally reconstructs the line parameters phase.

7.5.1.2.2 Definitions

t_max	Envelope parameter: time of maximum.
r_atk	Envelope parameter: attack rate.
r_dec	Envelope parameter: decay rate.
ampl[i]	Amplitude of i-th individual line.
freq[i]	Frequency of i-th individual line (in Hz).
startPhase[i]	Start phase of i-th individual line.
startPhaseAvail[i]	A flag indicating that start phase information is available for the i-th individual line.
t_maxEnh	Enhanced envelope parameter: time of maximum.
r_atkEnh	Enhanced envelope parameter: attack rate.
r_decEnh	Enhanced envelope parameter: decay rate.
amplEnh[i]	Enhanced amplitude of i-th individual line.
freqEnh[i]	Enhanced frequency of i-th individual line (in Hz).
phaseEnh[i]	Phase of i-th individual line (in rad).

7.5.1.2.3 Decoding process

7.5.1.2.3.1 Basic decoder

The basic decoder reconstructs the line parameters from the data contained in HILNbasicFrame() and INDIbasicPara() in the following way:

For each frame, first the number of individual lines encoded in this frame is read from HILNbasicFrame():

```
numLine
```

Then the frame envelope flag is read from HILNbasicFrame():

```
envFlag
```

If envFlag = 1 then the 3 envelope parameters t_max, r_atk, and r_dec are decoded from HILNbasicFrame():

```
t_max = (envTmax+0.5)/(1<<tmbits);
r_atk = tan(pi/2*max(0,envRatk-0.5)/((1<<atkbits)-1))/0.2;
r_dec = tan(pi/2*max(0,envRdec-0.5)/((1<<decbits)-1))/0.2;
```

These envelope parameters are valid for the harmonic lines as well as for the individual lines. Thus the envelope parameters envTmax, envRatk, envRdec must be dequantized if present, even if numLine == 0.

For each line k of the previous frame

```
k = 0 .. prevNumLine-1
```

the previous line continuation flag is read from HILNbasicFrame():

```
prevLineContFlag[k]
```

If prevLineContFlag[k] == 1 then line k of the previous frame is continued in the current frame. If prevLineContFlag[k] == 0 then line k of the previous frame is not continued.

In the current frame, first the parameters of all continued lines are encoded followed by the parameters of the new lines. Therefore, the line continuation flag and the line predecessor are determined before decoding the line parameters:

```
i = 0;
for (k = 0; k < prevNumLine; k++)
    if (prevLineContFlag[k]) {
        linePred[i] = k;
        lineContFlag[i++] = 1;
    }
while (i < numLine)
    lineContFlag[i++] = 0;
```

The parameters of new lines are encoded with increasing frequency index, using a differential encoding scheme. Therefore the following initialization is required once for each frame:

```
lastNLFreq = 0;
```

For each line i of the current frame

```
i = 0 .. numLine-1
```

the line parameters are decoded from INDIbasicPara() now.

If envFlag == 1 then the line envelope flag is read from INDIbasicPara():

```
lineEnvFlag[i]
```

If lineContFlag[i] == 1 then the parameters of a continued line are decoded from INDIbasicPara() based on the amplitude and frequency index of its predecessor in the previous frame:

```
ILFreqIndex[i] = prevILFreqIndex[linePred[i]] + DILFreq[i];
ILAmplIndex[i] = prevILAmplIndex[linePred[i]] + DILAmpl[i];
```

If lineContFlag[i] == 0 then the parameters of a new line are decoded from INDIbasicPara():

```
if (numLine-1-i < 7)
    ILFreqInc[i] = SDCdecode (maxFindex-lastNLFreq, sdcILFTable[numLine-1-i]);
else
    ILFreqInc[i] = SDCdecode (maxFindex-lastNLFreq, sdcILFTable[7]);

ILFreqIndex[i] = lastNLFreq + ILFreqInc[i];
lastNLFreq = ILFreqIndex[i];
if (HILNquantMode) {
    ILAmplRel[i] = SDCdecode (50, sdcILATable);
    ILAmplIndex[i] = maxAmplIndex + ILAmplRel[i];
}
```

```
else {
    ILAmplRel[i] = SDCdecode (25, sdcILATable);
    ILAmplIndex[i] = maxAmplIndex + 2*ILAmplRel[i];
}
```

The line parameter indices are stored for decoding of the line parameters of the next frame:

```
prevNumLine = numLine;
for (i = 0; i < prevNumLine; i++) {
    prevILFreqIndex[i] = ILFreqIndex[i];
    prevILAmplIndex[i] = ILAmplIndex[i];
}
```

The basic decoder also handles combinations of a basic layer and one or more extension layers. If data from a total of numLayer extension layers is available to the basic decoder, the values of layNumLine[numLayer] and layPrevNumLine[numLayer] are to be used instead of numLine and prevNumLine respectively. The values of ILAmplIndex[i], ILFreqIndex[i], lineContFlag[i], and linePred[i] as determined by the bitstream syntax description are to be used.

The amplitudes and frequencies of the individual lines are now dequantized from the indices:

```
for (i = 0; i < numLine; i++) {
    ampl[i] = 32768 * pow(10, -1.5*(ILAmplIndex+0.5)/20 );
    if ( ILFreqIndex<160 )
        freq[i] = (ILFreqIndex+0.5) * 3.125;
    else
        freq[i] = 500 * exp( 0.00625 * (ILFreqIndex+0.5-160) );
}
```

The optional start phase information is decoded as follows:

```
for (i = 0; i < numLine; i++) {
    if (linePhaseAvail[i]) {
        startPhase[i] = 2*pi*(linePhase[i]+0.5)/(1<<phasebits)-pi;
        startPhaseAvail[i] = 1;
    }
    else
        startPhaseAvail[i] = 0;
}
```

If the decoding process starts with an arbitrary frame of a bitstream, all individual lines which are marked in the bitstream as to be continued from previous frames which have not been decoded are to be muted.

7.5.1.2.3.2 Enhanced decoder

The enhanced decoder refines the line parameters obtained from the basic decoder and also decodes the line phases. The additional information is contained in bitstream element INDIenhaPara() and evaluated in the following way:

First, all operations of the basic decoder have to be carried out in order to allow correct decoding of parameters for continued lines.

If envFlag == 1 then the enhanced parameters t_maxEnh, r_atkEnh, and r_decEnh are decoded using the envelope data conveyed in HILNbasicFrame() and HILNenhaFrame():

```
t_maxEnh = (envTmax+(envTmaxEnh+0.5)/(1<<tmEnhbits))/(1<<tmbits);
if (envRatk == 0)
    r_atkEnh = 0;
else
    r_atkEnh = tan(pi/2*(envRatk-1+(envRatkEnh+0.5)/(1<<atkEnhbits))/
                  ((1<<atkbits)-1))/0.2;
if (envRdec == 0)
    r_decEnh = 0;
else
    r_decEnh = tan(pi/2*(envRdec-1+(envRdecEnh+0.5)/(1<<decEnhbits))/
```

```
                    ((1<<decbits)-1))/0.2;
```

For each line i of the current frame

```
   i = 0 .. numLine-1
```

the enhanced line parameters are obtained by refining the parameters from the basic decoder with the data in INDIenhaPara():

```
   amplEnh[i] = ampl[i];
   if (fEnhbits[i]!= 0) {
      if ( ILFreqIndex<160 )
         freqEnh[i] = (ILFreqIndex+0.5 + ((lineFreqEnh[i]+0.5)/(1<<fEnhbits[i])-0.5)) *
3.125;
      else
         freqEnh[i] = 500 * exp( 0.00625 * (ILFreqIndex+0.5-160 +
                     ((lineFreqEnh[i]+0.5)/(1<<fEnhbits[i])-0.5)) );
   }
   else
      freqEnh[i] = freq[i];
```

For both line types the phase is decoded from the enhancement bitstream:

```
   phaseEnh[i] = 2*pi*(linePhase[i]+0.5)/(1<<phasebits)-pi;
```

7.5.1.3 Noise decoder

7.5.1.3.1 Tool description

This tool decodes the noise parameters transmitted in the bitstream.

7.5.1.3.2 Definitions

prevNumNoisePara	The number of noise LPC parameters in the previous frame.
noiseLPCPara[i]	Noise LPC parameter i in the current frame (LARs for noise spectrum).
prevNoiseLPCPara[i]	Noise LPC parameter i in the previous frame (LARs for noise spectrum).
noiseAmpl	Noise amplitude.
noisePwr	Noise power.
noiseT_max	Noise envelope parameter: time of maximum.
noiseR_atk	Noise envelope parameter: attack rate.
noiseR_dec	Noise envelope parameter: decay rate.

7.5.1.3.3 Decoding process

7.5.1.3.3.1 Basic decoder

If the noiseFlag is set and thus NOISEbasicPara() data is available in the current frame, the parameters of the noise signal component are decoded and dequantized as follows:

The noise is represented by its power and a set of LPC-Parameters.

First the noiseNumPara LAR parameters are reconstructed. Prediction from the previous frame is used when noiseContFlag is set.

```
float noiseLPCMean[25] = { 2.0, -0.75, 0.0, 0.0, 0.0, ... , 0.0 };

for (i = 0; i < numNoisePara; i++) {
   if (i < prevNumNoisePara && noiseContFlag )
      pred = noiseLPCMean[i] + (prevNoiseLPCPara[i]-noiseLPCMean[i])*0.75;
   else
      pred = noiseLPCMean[i];
   noiseLPCPara[i] = pred + noiseLAR[i];
}
```

Parameters needed in the following frame are stored in the frame-to-frame memory:

```
prevNumNoisePara = numNoisePara;
for (i = 0; i < numNoisePara; i++) {
   prevNoiseLPCPara[i] = noiseLPCPara[i];
```

The amplitude and power of the noise is dequantized as follows:

```
noiseAmpl = 32768 * pow(10, -1.5*(noiseAmplIndex+0.5)/20 );
noisePwr = noiseAmpl*noiseAmpl;
```

If noiseEnvFlag == 1 then the noise envelope parameters noiseEnvTmax, noiseEnvRatk, and noiseEnvRdec are dequantized into noiseT_max, noiseR_atk, and noiseR_dec in the same way as described for the individual line decoder (see subclause 7.5.1.2.3.1).

7.5.1.3.3.2 Enhanced decoder

Since there is no enhancement data for noise components, there is no specific enhanced decoding mode for noise parameters. If noise is to be synthesized with enhancement data present for the other components, the basic noise parameter decoder can be used. However it has to be noted that if the HILN decoder is used as a core in a scalable coder no noise signal must be synthesized for the signal which is given to the enhancement decoder.

7.5.1.4 Harmonic and individual line synthesizer

7.5.1.4.1 Tool description

This tool synthesizes the audio signal according to the harmonic and individual line parameters decoded by the corresponding decoder tools. It includes the combination of the harmonic and individual lines, the basic synthesizer and the enhanced synthesizer. To obtain the complete decoded audio signal, the output signal of this tool is added to the output signal of the noise synthesizer as described in subclause 7.5.1.5.

7.5.1.4.2 Definitions

totalNumLine	Total number of lines in the current frame to be synthesized (individual plus harmonic).
sampleRate	Sampling rate in Hz as indicated by HILNsampleRateCode (see Table 7.7).
synthSampleRate	Sampling rate in Hz of synthesized output signal x[n].
speedFactor	Synthesis speed change factor (>1 for faster than original playback speed).
pitchFactor	Synthesis pitch change factor (>1 for higher than original playback pitch).
T	Synthesis frame length in seconds.
N	Synthesis frame length in samples.
env(t)	Amplitude envelope function in the current frame.
a(t)	Instantaneous amplitude of the line being synthesized.

p(t)	Instantaneous phase of the line being synthesized.
x(t)	Synthesized output signal.
x[n]	Sampled synthesized output signal.
startPhi[i]	Start phase of the i-th line in the current frame (in rad).
phi[i]	End phase of the i-th line in the current frame (in rad).
previousEnvFlag	Envelope flag in the previous frame.
previousT_max	Envelope parameter t_max in the previous frame.
previousR_atk	Envelope parameter r_atk in the previous frame.
previousR_dec	Envelope parameter r_dec in the previous frame.
previousEnv(t)	Amplitude envelope function in the previous frame.
previousTotalNumLine	Total number of lines in the previous frame.
previousAmpl[k]	Amplitude of the k-th line in the previous frame.
previousFreq[k]	Frequency of the k-th line in the previous frame (in Hz).
previousPhi[k]	End phase of the k-th line in the previous frame (in rad).
previousLineEnvFlag[k]	A flag indicating that the previous amplitude envelope is applied to the k-th line in the previous frame.
previousT_maxEnh	Enhanced envelope parameter t_max in the previous frame.
previousR_atkEnh	Enhanced envelope parameter r_atk in the previous frame.
previousR_decEnh	Enhanced envelope parameter r_dec in the previous frame.
previousAmplEnh[k]	Enhanced amplitude of the k-th line in the previous frame.
previousFreqEnh[k]	Enhanced frequency of the k-th line in the previous frame (in Hz).
previousPhaseEnh[k]	Phase of the k-th line in the previous frame (rad).

7.5.1.4.3 Synthesis process

7.5.1.4.3.1 Combination of harmonic and individual lines

For the synthesis of the harmonic lines the same synthesis technique as for the individual lines is used.

If no harmonic component is decoded for the following steps numHarmLine has to be set to zero.

Otherwise the parameters of the harmonic lines are appended to the list of individual line parameters as decoded by the individual line decoder:

```
for (i=0; i<numHarmLine; i++) {
   freq[numLine+i] = hLineFreq[i];
   ampl[numLine+i] = hLineAmpl[i];
   if (harmContFlag && prevNumLine+i < previousTotalNumLine) {
      lineContFlag[numLine+i] = 1;
```

```
            linePred[numLine+i] = prevNumLine+i;
        }
        else
            lineContFlag[numLine+i] = 0;
        lineEnvFlag[numLine+i] = harmEnvFlag;
        startPhase[numLine+i] = hStartPhase[i];
        startPhaseAvail[numLine+i] = hStartPhaseAvail[i];
    }
```

Thus the total number of line parameters passed to the harmonic and individual line synthesizer is:

```
totalNumLine = numLine + numHarmLine;
```

Depending on the value of HILNcontMode it is possible to connect lines in adjacent frames in order to avoid phase discontinuities in the case of transitions to and from harmonic lines (HILNcontMode == 0) or additionally from individual lines to individual lines for which the continue bit lineContFlag in the bitstream was not set by the encoder (HILNcontMode == 1). This additional line continuation as described below can also be completely disabled (HILNcontMode == 2).

For each line i = 0 .. totalNumLine-1 of the current frame that has no predecessor (i.e. lineContFlag[i] == 0), the best-fitting line j of the previous frame having no successor and with the combination meeting the requirements specified by HILNcontMode as described above is determined by maximizing the following measure q:

```
df = freq[i] / previousFreq[j];
df = max(df, 1/df);
da = ampl[i] / previousAmpl[j];
da = max(da, 1/da);
q = (1 - (df-1)/(dfCont-1)) * (1 - (da-1)/(daCont-1));
```

where dfCont = 1.05 and daCont = 4 are the maximum relative frequency and amplitude changes permitted. If there is more than one predecessor candidate achieving maximum q, the one with smallest index j is selected. For additional line continuations determined in this way, the line predecessor information is updated:

```
lineContFlag[i] = 1;
linePred[i] = j;
```

If there is not at least one possible predecessor with df < dfCont and da < daCont, lineContFlag[i] and linePred[i] remain unchanged.

For the enhanced synthesizer, the enhanced harmonic (up to maximum of 10) and individual line parameters are combined as follows:

```
for (i = 0; i < min(10,numHarmLine); i++) {
    freqEnh[numLine+i] = hLineFreqEnh[i];
    amplEnh[numLine+i] = hLineAmplEnh[i];
    if (harmContFlag && prevNumLine+i < previousTotalNumLine) {
        lineContFlag[numLine+i] = 1;
        linePred[numLine+i] = prevNumLine+i;
    }
    else
        lineContFlag[numLine+i] = 0;
    lineEnvFlag[numLine+i] = harmEnvFlag;
    phaseEnh[numLine+i] = hLinePhaseEnh[i];
}
```

Thus the total number of line parameters passed to the enhanced harmonic and individual line synthesizer, if the HILN decoder is used as a core in a scalable coder, is:

```
totalNumLine = numLine + min(10,numHarmLine);
```

Since phase information is available for all of these lines, no line continuation is introduced for the enhanced synthesizer.

7.5.1.4.3.2 Speed and pitch change

Due to the parametric signal representation utilized by the HILN coder and the phase continuation provided by the basic line synthesizer, the playback speed and pitch can be easily modified during the signal synthesis in the decoder. If playback at the original speed and pitch is desired, the corresponding control factors are set to their default values:

```
speedFactor = 1.0;
pitchFactor = 1.0;
```

If the speed is controlled by the time scaling factor in the **speed** field of the AudioSource BIFS node, the speed change factor is:

```
speedFactor = 1 / speed;
```

If the pitch is controlled by the **pitch** field of the AudioSource BIFS node, the pitch change factor is:

```
pitchFactor = pitch;
```

When the enhanced synthesizer is used instead of the basic synthesiser, speedFactor and pitchFactor should always be set to their default value 1.0.

Speed change is realized by modifying the synthesis frame length according to the desired speedFactor as described in subclause 7.5.1.4.3.3.

Pitch change is realized by modifying the frequency parameters of the harmonic and individual lines as follows:

```
for (i = 0; i < totalNumLine; i++) {
   freq[i] *= pitchFactor;
}
```

The noise synthesizer described in subclause 7.5.1.4.3.3 also supports speed and pitch change as detailed there.

7.5.1.4.3.3 Synthesis framing

The harmonic and individual line synthesizer reconstructs one frame of the audio signal. Since the line parameters encoded in a bitstream frame are valid for the center of the corresponding frame of the audio signal, the synthesizer generates the one-frame long section of the audio signal x(t) that starts at the center of the previous frame (t = 0) and that ends at the center of the current frame (t = T).

As default, the HILN synthesizer operates at the sampling frequency synthSampleRate as indicated by the samplingFrequency conveyed in the AudioSpecificInfo() (see subpart 1):

```
synthSampleRate = samplingFrequency
```

The synthesis frame contains N samples:

```
N = (int)(HILNframeLength * synthSampleRate / sampleRate / speedFactor + 0.5);
```

Thus the duration T of the synthesis frame is:

```
T = N / synthSampleRate;
```

In the following, the calculation of the synthesized output signal

```
x(t)
```

for 0 <= t < T is described. The time discrete version (i.e. the actual frame of output samples) is defined as

```
x[n] = x(t) with t = (n+0.5)*(T/N)
```

for 0 <= n < N.

The noise synthesizer described in subclause 7.5.1.5 uses the same synthesis framing as the harmonic and individual line synthesizer described here.

7.5.1.4.3.4 Basic synthesizer

Some parameters of the previous frame (names starting with “previous”) are taken out of the frame-to-frame memory which has to be reset before decoding the first frame of a bitstream.

First the envelope functions previousEnv(t) and env(t) of the previous and current frame are calculated according to the following rules:

If envFlag == 1 then the envelope function env(t) is derived from the envelope parameters t_max, r_atk, and r_dec. With T being the frame length, env(t) is calculated for -T/2 <= t < 3/2*T:

```
if (-1/2 <= t/T && t/T < t_max)
    env(t) = max(0,1-(t_max-t/T)*r_atk);
if (t_max <= t/T && t/T < 3/2)
    env(t) = max(0,1-(t/T-t_max)*r_dec);
```

If envFlag == 0 then a constant envelope function env(t) is used:

```
env(t) = 1;
```

Accordingly previousEnv(t) is calculated from the parameters previousT_max, previousR_atk, previousR_dec and previousEnvFlag.

The envelope parameters transmitted in case of envFlag == 1 are valid for the harmonic lines as well as for the individual lines. Thus the envelope functions always must be generated, even if all lineEnvFlag[i] == 0.

Before the synthesis is performed, the accumulator x(t) for the synthesized audio signal is cleared for 0 <= t < T:

```
x(t) = 0;
```

The lines i continuing from the previous frame to the current frame

```
all i = 0 .. totalNumLine-1 that have lineContFlag[i] == 1
```

are synthesized as follows for 0 <= t < T:

```
k = linePred[i];
ap(t) = previousAmpl[k];
if (previousLineEnvFlag[k] == 1)
    ap(t) *= previousEnv(t+T/2);
ac(t) = ampl[k];
if (lineEnvFlag[i] == 1)
    ac(t) *= env(t-T/2);

short_x_fade = (previousLineEnvFlag[k] && !(previousR_dec < 5 &&
                    (previousT_max < 0.5 || previousR_atk < 5))) ||
                    (lineEnvFlag[i] && !(r_atk < 5 && (t_max > 0.5 || r_dec < 5)));

if (short_x_fade == 1) {
    if (0 <= t && t < 7/16*T)
        a(t) = ap(t);
    if (7/16*T <= t && t < 9/16*T)
        a(t) = ap(t) + (ac(t)-ap(t))*(t/T-7/16)*8;
    if (9/16*T <= t&& t < T)
        a(t) = ac(t);
}
else
```

```
    a(t) = ap(t) + (ac(t)-ap(t))*t/T;
p(t) = previousPhi[k]+2*pi*previousFreq[k]*t+
                2*pi*(freq[i]-previousFreq[k])/(2*T)*t*t;
x(t) += a(t)*sin(p(t));
phi[i] = p(T)
```

The lines i starting in the current frame

```
all i = 0 .. totalNumLine-1 that have lineContFlag[i] == 0
```

are synthesized as follows for 0 <= t < T:

```
if (lineEnvFlag[i] && !(r_atk < 5 && (t_max > 0.5 || r_dec < 5))) {
    if (0 <= t&& t < 7/16*T)
        fade_in(t) = 0;
    if (7/16*T <= t&& t < 9/16*T)
        fade_in(t) = 0.5 - 0.5*cos((8*t/T-7/2)*pi);
    if (9/16*T <= t&& t < T)
        fade_in(t) = 1;
}
else
    fade_in(t) = 0.5-0.5*cos(t/T*pi);
a(t) = fade_in(t)*ampl[i];
if (lineEnvFlag[i] == 1)
    a(t) *= env(t-T/2);
if (startPhaseAvail[i])
    startPhi[i] = startPhase[i];
else
    startPhi[i] = random(2*pi);
p(t) = startPhi[i] + 2*pi*freq[i]*(t-T);
x(t) += a(t)*sin(p(t));
phi[i] = p(T)
```

random(x) is a function returning a random number with uniform distribution in the interval

```
0 <= random(x) < x
```

The lines k ending in the previous frame

```
all k = 0 .. previousTotalNumLine-1 that have prevLineContFlag[k] == 0
```

are synthesized as follows for 0 <= t < T:

```
if (previousLineEnvFlag[k] && !(previousR_dec < 5 &&
        (previousT_max < 0.5 || previousR_atk < 5))) {
    if (0 <= t&& t < 7/16*T)
        fade_out(t) = 1;
    if (7/16*T <= t&& t < 9/16*T)
        fade_out(t) = 0.5 + 0.5*cos((8*t/T-7/2)*pi);
    if (9/16*T <= t&& t < T)
        fade_out(t) = 0;
}
else
    fade_out(t) = 0.5+0.5*cos(t/T*pi);
a(t) = fade_out(t)*previousAmpl[k];
if (previousLineEnvFlag[k] == 1)
    a(t) *= previousEnv(t+T/2);
p(t) = previousPhi[k]+2*pi*previousFreq[k]*t;
x(t) += a(t)*sin(p(t));
```

In order to avoid aliasing distortion, synthesized lines are muted (i.e. a(t) = 0) while their instantaneous frequency is above or equal half the sampling frequency, i.e.

```
d phi(t) / dt >= pi*N/T
```

Parameters needed in the following frame are stored in the frame-to-frame memory:

```
previousEnvFlag = envFlag;
previousT_max = t_max;
previousR_atk = r_atk;
previousR_dec = r_dec;
previousTotalNumLine = totalNumLine;
for (i=0; i<totalNumLine; i++) {
    previousFreq[i] = freq[i];
    previousAmpl[i] = ampl[i];
    previousPhi[i] = fmod(p[i],2*pi);
    previousLineEnvFlag[i] = lineEnvFlag[i];
}
```

fmod(x,2*pi) is a function returning the 2*pi modulus of x.

7.5.1.4.3.5 Enhanced synthesizer

The enhanced synthesizer is based on the basic synthesizer but evaluates also the line phases for reconstructing one frame of the audio signal. Since the line parameters encoded in a bitstream frame and the corresponding enhancement frame are valid for the middle of the corresponding frame of the audio signal, the harmonic and individual line synthesizer generates the one frame long section of the audio signal that starts in the middle of the previous frame and ends in the middle of the current frame.

Some parameters of the previous frame (names starting with “previous”) are taken out of the frame-to-frame memory which has to be reset before decoding the first frame of a bitstream.

First the envelope functions previousEnv(t) and env(t) of the previous and current frame are calculated according to the following rules:

If envFlag == 1 then the envelope function env(t) is derived from the envelope parameters t_maxEnh, r_atkEnh, and r_decEnh. With T being the frame length, env(t) is calculated for -T/2 <= t < 3/2*T:

```
if (-1/2 <= t/T && t/T < t_maxEnh)
    env(t) = max(0,1-(t_maxEnh-t/T)*r_atkEnh);
if (t_maxEnh <= t/T && t/T < 3/2)
    env(t) = max(0,1-(t/T-t_maxEnh)*r_decEnh);
```

If envFlag == 0 then a constant envelope function env(t) is used:

```
env(t) = 1;
```

Accordingly previousEnv(t) is calculated from the parameters previousT_maxEnh, previousR_atkEnh, previousR_decEnh and previousEnvFlag.

The envelope parameters transmitted in case of envFlag == 1 are valid for the harmonic lines as well as for the individual lines. Thus the envelope functions always must be generated, even if all lineEnvFlag[i] == 0.

Before the synthesis is performed, the accumulator x(t) for the synthesized audio signal is cleared for 0 <= t < T:

```
x(t) = 0
```

All lines i in the in the current frame

```
all i =0 .. totalNumLine-1
```

are synthesized as follows for 0 <= t < T:

```
if (envFlag && !(r_decEnh < 5 && (t_maxEnh < 0.5 || r_atkEnh < 5))) {
    if (0 <= t&& t < 7/16*T)
        fade_in(t) = 0;
    if (7/16*T <= t&& t < 9/16*T)
        fade_in(t) = 0.5 - 0.5*cos((8*t/T-7/2)*pi);
```

```
    if (9/16*T <= t&& t < T)
        fade_in(t) = 1;
}
else
    fade_in(t) = 0.5-0.5*cos(t/T*pi);
a(t) = fade_in(t)*amplEnh[i];
if (envFlag[i] == 1)
    a(t) *= env(t-T/2);
phi(t) = 2*pi*freqEnh[i]*(t-T)+phaseEnh[i];
x(t) += a(t)*sin(phi(t));
```

The lines k in the previous frame

```
all k=0 .. previousTotalNumLine-1
```

are synthesized as follows for 0 <= t < T:

```
if (previousEnvFlag && !(previousR_atkEnh < 5 &&
        (previousT_maxEnh > 0.5 || previousR_decEnh < 5))) {
    if (0 <= t&& t < 7/16*T)
        fade_out(t) = 1;
    if (7/16*T <= t&& t < 9/16*T)
        fade_out(t) = 0.5 + 0.5*cos((8*t/T-7/2)*pi);
    if (9/16*T <= t&& t < T)
        fade_out(t) = 0;
}
else
    fade_out(t) = 0.5+0.5*cos(t/T*pi);
a(t) = fade_out(t)*previousAmplEnh[k];
if (previousEnvFlag[k] == 1)
    a(t) *= previousEnv(t+T/2);
phi(t) = 2*pi*previousFreqEnh[k]*t+previousPhaseEnh[i];
x(t) += a(t)*sin(phi(t));
```

If the instantaneous frequency of a line is above or equal half the sampling frequency, i.e.

```
d phi(t) / dt >= pi*N/T
```

it is not synthesized to avoid aliasing distortion.

Parameters needed in the following frame are stored in the frame-to-frame memory:

```
previousEnvFlag = envFlag;
previousT_maxEnh = t_maxEnh;
previousR_atkEnh = r_atkEnh;
previousR_decEnh = r_decEnh;
previousTotalNumLine = totalNumLine;
for (i=0; i<totalNumLine; i++) {
    previousFreqEnh[i] = freqEnh[i];
    previousAmplEnh [i] = amplEnh[i];
    previousPhaseEnh [i] = phaseEnh [i];
}
```

7.5.1.5 Noise synthesizer

7.5.1.5.1 Tool description

This tool synthesizes the noise part of the output signal based on the noise parameters decoded by the noise decoder. Finally the noise signal is added to the output signal of the harmonic and individual line synthesizer (subclause 7.5.1.4) to obtain the complete decoded audio signal

7.5.1.5.2 Definitions

noiseWin[n] Window for noise overlap-add.

noiseEnv[n]	Envelope for noise component in the current frame.
M	Frame length in samples before resampling.
w[m]	White noise with power pw.
xf[m]	Filtered noise signal in the current frame.
xn[n]	Synthesized noise signal in the current frame.
previousXn[n]	Synthesized noise signal in the previous frame.
previousNoiseWin[n]	Window and envelope for noise component in the previous frame.
r[i]	LPC reflection coefficients.
h[i]	LPC impulse response.
hlp[i]	Low-pass resampling filter impulse response.

7.5.1.5.3 Synthesis process

7.5.1.5.3.1 Basic synthesizer

If noise parameters are transmitted for the current frame, a noise signal with a spectral shape as described by the noise parameters decoded from the bitstream is synthesized and added to the audio signal generated by the harmonic and individual line synthesizer. The synthesis framing is as described in subclause 7.5.1.4.3.3.

The noise is represented by its power and a set of LPC-Parameters. As described in the harmonic tone decoder (subclause 7.5.1.1.3.1), the noise LPC parameters are converted to the reflection coefficients r[i] and to the time response h[i]:

```
for (i = 0; i < numNoisePara; i++)
   r[i] = ( exp(noiseLPCPara[i]) - 1 ) / ( exp(noiseLPCPara[i]) + 1 );
```

After this the reflection coefficients r[i] are converted to the time response h[i] using the C function

```
void Convert_k_to_h (float *x, int N)
```

given in subclause 7.5.1.1.3.1 (call with x[i] = r[i] and N = numNoisePara; returns with x[i] = h[i]).

Now the filtered noise signal xf[m] is generated by applying the LPC synthesis IIR filter to a white noise represented by random numbers w[m]. The power of this zero-mean white noise is denoted pw. For a noise with uniform distribution in [-1,1] the power is

```
pw = 1/3
```

To achieve the required noise signal power, the following scaling factor s is required:

```
ss = 1.0;
for (i = 0; i < numNoisePara; i++)
   ss *= 1-r[i]*r[i];
s = noiseAmpl * sqrt( ss/pw );
```

Then the white noise w[m] is IIR-filtered to obtain the synthesized noise signal xf[m]

```
for (m = startup; m < 2*M; m++) {
   xf[m] = s * w[m];
   for (i = 0; i < min(m-startup,numNoisePara); i++)
      xf[m] += h[i]*xf[m-i-1];
}
```

To ensure that the IIR filter can reach a sufficiently steady state, a startup phase is used:

```
startup = -numNoisePara
```

If decoded with a pitch change (i.e. pitchPactor != 1.0, see subclause 7.5.1.4.3.2) or with a different sample rate than the encoder (i.e. synthSampleRate != sampleRate), a resample operation must be applied to the signal xf[m] using the resampling factor

```
resampleFactor = ( sampleRate * pitchFactor ) / synthSampleRate;
```

where e.g. pitchFactor of 2 indicates that this signal is synthesized at twice its original pitch. Otherwise the resampleFactor is set to 1.0. Based on the resampleFactor, the framle length M before resampling is defined:

```
M = N * resampleFactor;
```

The resampling can be realized by applying two lowpass FIR filter operations to the signal xf[m] and linearly interpolating between these two values to obtain the final noise signal xn[n].

```
if (resampleFactor < 1)
    fc = 1;
else
    fc = 1/resampleFactor;
```

The following function calculates the time response hlp[0..31] of an appropriate lowpass FIR filter with 16 taps and an oversampling factor of 4. The cutoff frequency is fc.

```
void GenLPFilter (float *hlp, double fc)
{
    double x,f;
    int i;

    hlp[0] = (float) fc;
    for (i = 1; i < 32; i++)
    {
        x = i*pi/4.0;
        hlp[i] = (float) ((0.54+0.46*cos(0.125*x))*sin(fc*x)/x);
    }
}
```

To perform the FIR filter operation the following C function can be used. The parameters are the signal, the time response (as returned by the function above) and the position of the sampling point. The position is given as the difference between the nearest sample position prior to the desired sample position (x[7]) and the desired sample position. Therefore 0 <= pos < 1. The interpolation is done between x[7] and x[8], the returned value represents a sample position of 7+pos.

```
float LPInterpolate (float *x, float *hlp, double pos)
{
    long   j;
    double s,t;

    pos *= 4.0;
    j = (long) pos;
    pos -= (double) j;

    s = t = 0.0;
    j = 32-j;

    if (j == 32)
    {
        t = h[31]*(*x);
        x++;
        j -= 4;
    }
    while (j > 0)
    {
        s += h[j ]*(*x);
```

```
        t += h[j-1]*(*x);
        x++;
        j -= 4;
    }
    j = -j;
    while (j < 32-1)
    {
        s += h[j ]*(*x);
        t += h[j+1]*(*x);
        x++;
        j += 4;
    }
    if (j < 32)
        s += h[j]*(*x);

    return (float) (s+pos*(t-s));
}
```

Using the functions GenLPFilter() and LPInterpolate(), the resampling is done as described below. xf[m] is set to 0.0 for m < startup and m >= 2*M.

```
GenLPFilter(hlp, fc);
for (n = 0; n < 2*N; n++)
    xn[n] = LPInterpolate(xf+((int)n*resampleFactor)-7, hlp, frac(n*resampleFactor));
```

If resampleFactor == 1.0, xf[m] is simply copied to xn[n] without resampling:

```
for (n = 0; n < 2*N; n++)
    xn[n] = xf[n];
```

For smooth cross-fade of the noise signal at the boundary between two adjacent frames, the following window is used for this overlap-add operation:

```
for (n = 0; n < N; n++) {
    if (n < N*3/8)
        noiseWin[n] = 0;
    if (N*3/8 <= n && n < N*5/8)
        noiseWin[n] = sin(pi/2 * (n-N*3/8+0.5)/(N*2/8));
    if (N*5/8 <= n)
        noiseWin[n] = 1;
    noiseWin[2*N-1-n] = noiseWin[n];
}
```

Now the envelope function noiseEnv[n] is calculated. If noiseEnvFlag == 1 then the envelope function

```
noiseEnv[n] = noiseEnv(t) with t = (n+0.5)*(T/N)-0.5
```

is derived from the envelope parameters noiseT_max, noiseR_atk, and noiseR_dec for -T/2 <= t < 3/2*T (i.e. 0 <= n < 2*N):

```
if (-1/2 <= t/T && t/T < noiseT_max)
    noiseEnv(t) = max(0,1-(noiseT_max-t/T)*noiseR_atk);
if (noiseT_max <= t/T && t/T < 3/2)
    noiseEnv(t) = max(0,1-(t/T-noiseT_max)*noiseR_dec);
```

If noiseEnvFlag == 0 then a constant envelope function noiseEnv(t) is used:

```
noiseEnv[n] = 1;
```

The noise signal xn[n] is windowed for overlap-add and multiplied with the envelope noiseEnv[n]. Then this signal and the noise from the previous frame previousXn[n] are added to the signal x[n] from the harmonic and individual line synthesizer to construct the complete synthesized signalx[n]:

```
for (n = 0; n < N; n++)
    x[n] += xn[n]*noiseWin[n]*noiseEnv[n] + previousXn[n];
```

The second half of the generated noise signalxn[n] is stored in the frame-to-frame memory previousXn[n] for overlap-add:

```
for (n = 0; n < N; n++)
   previousXn[n] = xn[N+n]*noiseWin[N+n]*noiseEnv[N+n];
```

The previousXn[n] memory has to be reset to 0.0 before decoding of the first frame.

7.5.1.5.3.2 Enhanced synthesizer

Since there is no enhancement data for noise components, there is no specific enhanced synthesizer mode for noise components. If noise is to be synthesized with enhancement data present for the other components, the basic noise synthesizer decoder can be used. However it has to be noted that if the HILN decoder is used as a core in a scalable coder no noise signal must be synthesized for the signal which is given to the enhancement decoder.

7.5.2 Integrated parametric coder

The integrated parametric coder can operate in four different modes as shown in Table 7.1. PARAmodes 0 and 1 represent the fixed HVXC and HILN modes. PARAmode 2 permits automatic switching between HVXC and HILN depending on the current input signal type. In PARAmode 3 the HVXC and HILN coders can be used simultaneously and their output signals are added (mixed) in the decoder.

The integrated parametric coder uses a frame length of 40 ms and a sampling rate of 8 kHz and can operate at 2025 bit/s or any higher bitrate. Operation at 4 kbit/s or higher is suggested.

7.5.2.1 Integrated parametric decoder

For the “HVXC only” and “HILN only” modes the parametric decoder is not modified.

In “switched HVXC / HILN” and “mixed HVXC / HILN” modes both HVXC and HILN decoder tools are operated alternatively or simultaneously according to the PARAswitchMode or PARAmixMode of the current frame. To obtain proper time alignment of both HVXC and HILN decoder output signals before they are added, the difference between HVXC and HILN decoder delay has to be compensated with a FIFO buffer:

- If HVXC is used in the low delay decoder mode, its output must be delayed for 100 samples (i.e. 12.5 ms).
- If HVXC is used in the normal delay decoder mode, its output must be delayed for 80 samples (i.e. 10 ms).

To avoid hard transitions at frame boundaries when the HVXC or HILN decoders are switched on or off, the respective decoder output signals are faded in and out smoothly. For the HVXC decoder a 20 ms linear fade is applied when it is switched on or off. The HILN decoder requires no additional fading because of the smooth synthesis windows utilized in the HILN synthesizer. It is only necessary to operate the HILN decoder with no new components for the current frame (i.e. force numLine = 0, harmFlag = 0, noiseFlag = 0) if the current bitstream frame contains no “HILNframe()”.

7.6 Error resilient bitstream payloads

7.6.1 Overview of the tools

Error resilient bitstream payloads allow the effective usage of advanced channel coding techniques like unequal error protection (UEP), which can be perfectly adapted to the needs of the different coding tools. The basic idea is to rearrange the conventional bitstream payload depending on its error sensitivity in one or more instances belonging to different error sensitivity categories (ESC). This re-arrangement works either data element-wise or even bit-wise. The error resilient bitstream payload is built by concatenating these instances.

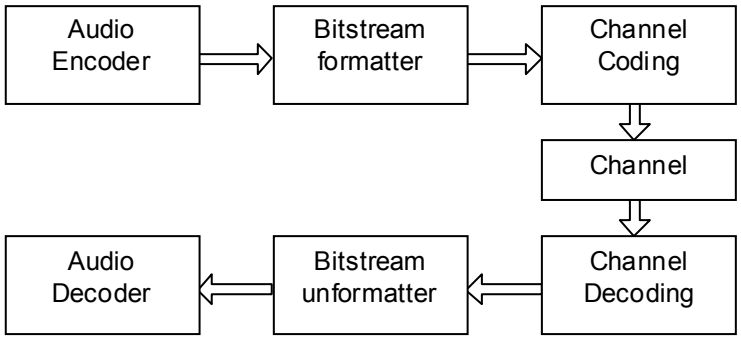


Figure 7.2 — Basic principle of error resilient bitstream reordering

The basic principle is depicted in Figure 7.2. A bitstream is reordered according to the error sensitivity of single bitstream elements or even single bits. This new arranged bitstream is channel coded, transmitted and channel decoded. Prior decoding, the bitstream is rearranged to its original order. Instead of performing the reordering in the described way, the reordered syntax, that is the bitstream order prior the bitstream formatter at the decoder site, is defined in this amendment.

In the subsequent subparts, a detailed description of error resilient bitstream reordering for these tools can be found.

7.6.2 ER HILN

Error sensitivity categories (ESC) are defined in the parametric bitstream as described in subclause 7.3. Below, the ordering of the different ESCx is described for the four different modes PARAmode == 0, 1, 2, 3.

- **PARAmode == 0 (HVXC only)**
  HVXC: ESC0 ESC1 ESC2 ESC3 ESC4

- **PARAmode == 1 (HILN only)**
  PARA/HILN: ESC0 ESC1 ESC2 ESC3 ESC4

- **PARAmode == 2 (switched HVXC / HILN)**
  PARA/HILN: ESC0 [ESC1 ESC2 ESC3 ESC4]
  HVXC 1/double: [ESC0 ESC1 ESC2 ESC3 ESC4]
  HVXC 2/double: [ESC0 ESC1 ESC2 ESC3 ESC4]

- **PARAmode == 3 (mixed HVXC / HILN)**
  PARA/HILN: ESC0 [ESC1 ESC2 ESC3 ESC4]
  HVXC 1/double: [ESC0 ESC1 ESC2 ESC3 ESC4]
  HVXC 2/double: [ESC0 ESC1 ESC2 ESC3 ESC4]

In PARAmode 3 for example, the ordering is PARA/HILN ESC0, ESC1, …, ESC4, followed by HVXC 1/double ESC0, ESC1, …, ESC4 and HVXC 2/double ESC0, ESC1, …, ESC4.

The ESC0 of “PARA/HILN” consists of the PARAswitchMode or PARAmixMode bitstream element in PARAframe() followed by the bitstream elements in HILNbasicFrameESC0(). The actual presence of these bitstream elements can depend on the current values of PARAmode, PARAswitchMode, and PARAmixMode. “HVXC 1/double” and “HVXC 2/double” denote the first and second ErHVXCfixframe() within an ErHVXCdoubleframe() respectively. The presence of ESCs in squared brackets depends on the value of PARAswitchMode or PARAmixMode in the current frame.

For the HILN enhancement and extension layer Elementary Streams, no error sensitivity categories are defined and all bitstream elements of HILNenexFrame() are handled as “ESC0”.

Annex 7.A
(informative)

Parametric audio encoder

7.A.1 Overview of the encoder tools

The following Figure 7.A.1 shows a general block diagram of a parametric encoder. First the input signal is separated into the two parts which are coded by HVXC and by HILN tools. This can be done manually or automatically. Automatic switching between speech and music signals is supported (see subclause 7.A.3.1), allowing the use of HVXC for speech and HILN for music. For both HVXC and HILN parameter estimation and parameter encoding can be performed. A common bitstream formatter allows operation either in HVXC only, HILN only, or also in combined modes, i.e. switched or mixed mode.

The following description of an HILN parametric encoder is informative and also alternative techniques for signal separation and parameter estimation can be used in an encoder.

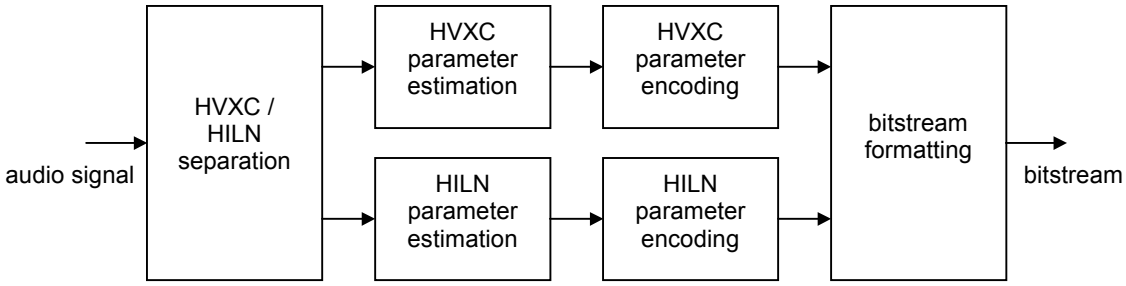


Figure 7.A.1 — General block diagram of the integrated parametric encoder

7.A.2 HILN encoder tools

The basic principle of the “Harmonic and Individual Lines plus Noise” (HILN) encoder is to analyze the input signal in order to extract parameters describing the signal. These parameters are coded and transmitted as a bitstream. In the decoder the output signal is synthesized based on the parameters extracted and transmitted by the encoder.

The encoder consists of two main parts: “Parameter Extraction” and “Parameter Coding”. In the encoder, the input signal is divided into consecutive frames and for each frame a set of parameters describing the signal in this frame is extracted and coded. Due to this parametric description, a wide range of bitrates, sampling rates and frame lengths are possible. Typically a frame length of 32 ms is used. For input signals with 8 to 16 kHz sampling rate typically a bitrate of 6 to 16 kbit/s is used.

The “Parameter Extraction” and “Parameter Coding” is described in detail in the following Subclauses.

7.A.2.1 HILN parameter extraction

Since different parameter sets and different synthesis techniques can be applied, the input signal of the encoder has to be split up in an appropriate way. This is performed by the *Separation* unit. Depending on the most appropriate synthesis technique, a parameter set is derived for each part of the input signal in the *Model Based Parameter Estimation* unit. The two units *Separation* and *Model Based Parameter Estimation* can be regarded as the analysis stage which produces a parametric description of the input signal. The separation of the input signal is

enhanced by feeding back the signals which are generated in the *Synthesis* unit from all the parameters of previously separated parts. The *Separation* and the *Model Based Parameter Estimation* additionally receive data from a synthesis model independent *Pre-Analysis*. Prior to transmission, the parameters are fed through the *Quantization and Coding* unit, which is controlled by a *Psychoacoustic Model*. This *Psychoacoustic Model* processes the input signal in order to derive information about the relevancy of synthesis parameters. In addition, the synthesized signal is fed into the *Psychoacoustic Model*, which thus is allowed to assist the *Model Based Parameter Estimation*.

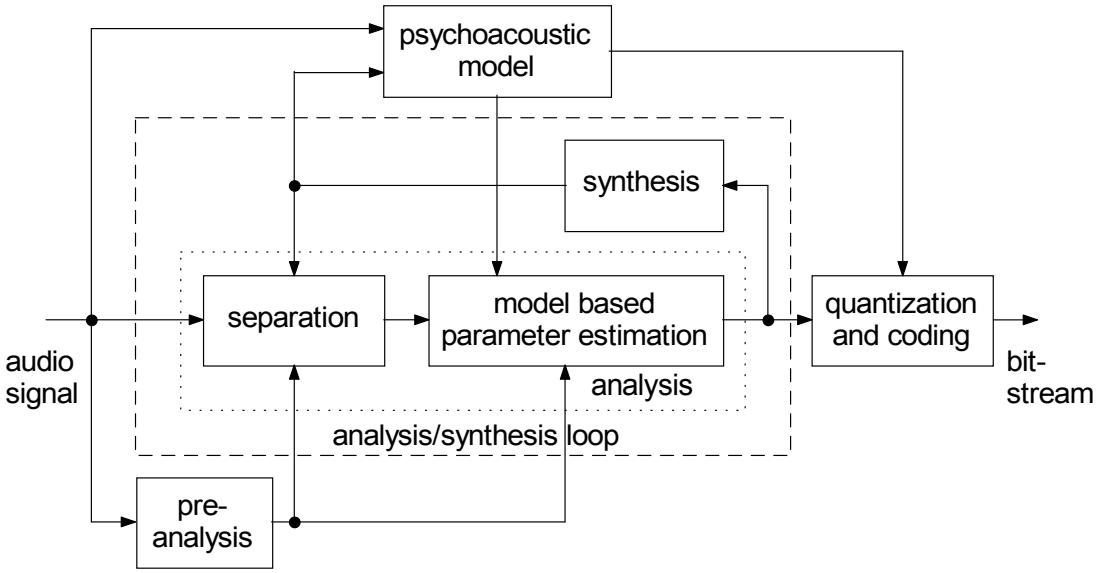


Figure 7.A.2 — Block diagram of the HILN encoder

In the parameter extraction, the input signal is separated into three different parts: “harmonic lines”, “individual lines” and “noise”.

For each of these parts parameters describing the signal are extracted. These are basically:

- harmonic lines: fundamental frequency and amplitudes of the harmonic components
- individual lines: frequency and amplitude of each individual line
- noise: spectral shape of the noise

Additionally parameters for amplitude envelopes and for continuation of spectral lines from one frame to the next can be determined.

The signal separation and parameter estimation is implemented in three steps: First the fundamental frequency of the harmonic part of the signal is estimated. Then the parameters of the relevant spectral lines are estimated and these lines are classified as “individual lines” or “harmonic lines” depending on the frequency with respect to the fundamental frequency. After all relevant spectral lines are extracted, the remaining residual signal is assumed to be noise-like and its spectral shape is described by a set of parameters.

The harmonic line extraction of the HILN tools can also be utilized in an integrated parametric coder utilizing both the HVXC speech coding tools as well as the HILN coding tools simultaneously. If the input signal is e.g. a speech signal mixed with background music, the HILN encoder can be used to extract only those individual spectral lines that do not belong to the harmonic part of the signal. These individual lines are encoded by the HILN tools and the remaining signal - consisting of the harmonic signal part and noise - then is encoded by the HVXC parametric speech codec tools. In the decoder, the audio signal is reconstructed by adding the output of the “individual line” synthesizer and the HVXC decoder.

7.A.2.1.1 Fundamental frequency estimation

A “cepstrum”-based fundamental frequency estimation technique is employed by the HILN tools. First the input signal is windowed with a Hanning window of twice the frame length centered around the current frame. For the windowed signal, the magnitude spectrum is calculated and the logarithm is applied to the magnitude spectrum. Then the log spectrum is multiplied with the window

```
w(f) = (1+cos(2*pi*f/fs))/2          0 <= f <= fs/2
```

and zero-padding is used to virtually double the sampling frequency before the cepstrum is calculated. Finally the local maxima in the cepstrum are determined and the largest maximum within the permitted “pitch lag” search range is identified. The fundamental frequency is calculated from the “pitch lag” (period of the fundamental frequency) of the largest maximum.

The fundamental frequency determined by this cepstrum-based technique is used as an initial (coarse) estimate in the following line parameter estimation.

7.A.2.1.2 Harmonic and Individual Line Parameter Estimation

The estimation of harmonic and individual line parameters is based on an “Analysis/Synthesis Loop” described in the following Subclauses.

In a first step the parameters of all harmonic lines are estimated. This is done by performing the regression-based high accuracy frequency estimation described below for all integer multiples of the coarse fundamental frequency as initial estimates. Based on the accurate frequencies of the harmonic lines, a fine estimate of the fundamental frequency hFreq and the so-called “stretching” hStretch is calculated which minimizes the total error between the real harmonic line frequencies and those calculated according to

```
hLinefreq[i] = hFreq * (i+1) * (1 + hStretch*(i+1))          i = 0 .. harmNumLine-1
```

where the total number of harmonic lines is determined by the bandwidth w of the signal and the current fundamental frequency hFreq:

```
harmNumLine = floor(w/hFreq)
```

The harmonic envelope flag is set if using the current amplitude envelope for all harmonic lines results in a lower residual error than if no envelope is used. If the relative change of the fundamental frequency between the previous and the current frame is less than 15%, the harmonic continuation flag is set.

In the second step the relevant spectral lines are extracted from the input signal by means of the “Analysis/Synthesis Loop”. This loop utilizes a psychoacoustic model to extract the spectral lines in order of their subjective relevance. If the frequency of an extracted spectral line is close to the frequency of a harmonic line as calculated from hFreq and hStretch, this extracted line is classified as harmonic line. Otherwise it is classified as individual line. The “Analysis/Synthesis Loop” is terminated if the requested number of individual lines was extracted or if the remaining signal components cannot be properly modeled by spectral lines. The ratio between the number of harmonic lines extracted and the total lines extracted is passed on to the encoder as measure of “relevance” of the harmonic lines.

If less than three extracted lines were classified as “harmonic line”, these lines are added to the list of “individual lines” and numHarmLine is set to 0. Finally all harmonic lines that were not extracted by the “Analysis/Synthesis Loop” are also removed from the residual signal. This residual signal is then passed to the noise parameter estimation.

7.A.2.1.2.1 Pre-Analysis

The pre-analysis module determines the signal amplitude envelope which is used in the analysis/synthesis loop.

7.A.2.1.2.2 Analysis/Synthesis Based on Single Spectral Lines

The Individual Line encoder is based on the model of single spectral lines, which can be generated with the help of sine wave generators. The according *Model Based Parameter Estimation* for the i-th line in the loop consists of the following steps:

- calculation of the deviation between FFT spectra of input and synthesized signals
- selection of the most relevant FFT line with center frequency $f_{i,m}$
- high resolution frequency estimation in the surrounding of $f_{i,m}$
- amplitude and phase estimation, selection of envelope information
- synthesis with the determined parameters
- calculation of the residual error signal by subtraction of the synthesized signal from the input signal

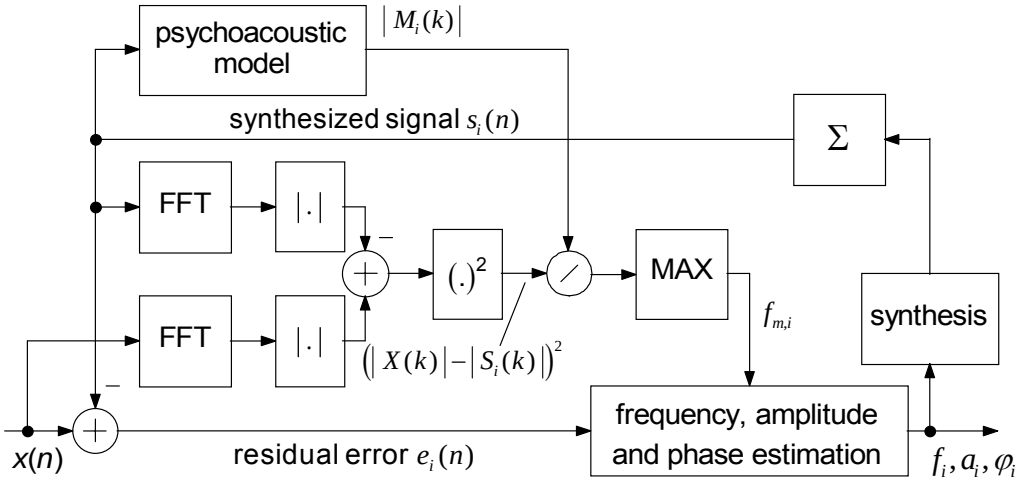


Figure 7.A.3 — Analysis/Synthesis Loop based on the synthesis method "single spectral lines"

The FFT line to be processed is determined by calculating the deviation between input spectrum and synthesized spectrum and searching the maximum ratio of the square of this deviation and the masking threshold derived from the signal synthesized from the previously determined spectral lines.

Based on the center frequency $f_{i,m}$ of the selected FFT line a frequency estimation is performed in order to obtain a frequency parameter of higher accuracy than the FFT resolution.

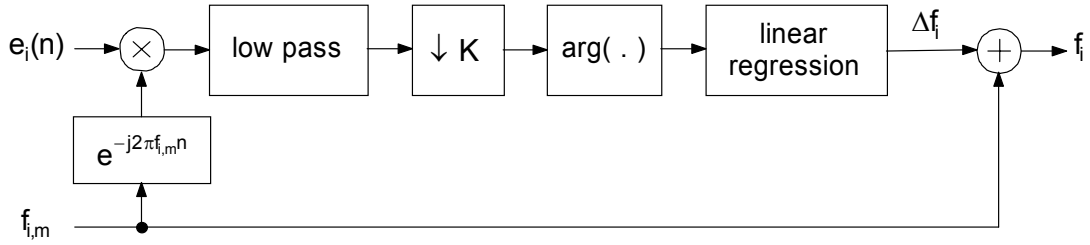


Figure 7.A.4 — High accuracy frequency estimation

For this frequency estimation, first the spectrum of the residual error signal is shifted in a way that the center frequency $f_{i,m}$ of the selected FFT line becomes zero. The complex output values of this operation are fed into lowpass filter and sampling rate reduction, which are realized by applying time-shifted versions of the window function. The slope of the regression line for the phase values of the obtained complex samples gives a frequency offset, which is added to $f_{i,m}$ in order to give a high resolution frequency parameter f_i. The time shift of the window function ranges from -0.32 to 0.32 times the frame length. The time shift step size is 0.08 times the frame length and thus 9 data points are used for the linear regression.

Based on f_i the amplitude and phase of the spectral line are calculated. This is realized by calculating a complex correlation coefficient of the residual error signal and a complex harmonic signal of frequency f_i. The absolute value of the correlation coefficient gives the amplitude parameter a_i and the argument gives the phase parameter φ_i. If the *Pre-Analysis* has generated envelope parameters, a second set of parameters $a_{i,e}$ and $\varphi_{i,e}$ is generated by correlation of the residual error signal and a complex harmonic signal of frequency f_i multiplied with the according envelope.

In the *Separation* unit, a new residual error signal is generated by subtracting the signal synthesized from the parameters f_i, a_i and φ_i. If the parameters $a_{i,e}$ and $\varphi_{i,e}$ are also available, a second signal is synthesized accordingly and the parameter set leading to the lowest residual error variance is selected.

7.A.2.1.2.3 Psychoacoustic model

The psychoacoustic model calculates the masked threshold for the synthesized signal components in the analysis/synthesis loop.

7.A.2.1.3 Noise Parameter Estimation

The noise parameters are used to model the spectral shape of the residual signal. First the power spectrum of the Hanning-windowed residual signal is calculated. Then this spectrum is transformed back into the autocorrelation function. Based on this autocorrelation function, LPC parameters are calculated using Durbin's algorithm. These LPC parameters are then transformed into reflection coefficients which are represented as Log Area Ratios (LAR). Besides the LAR parameters also the power of the noise signal is calulated.

Additionally a new set of envelope parameters is calculated for the residual signal. Thus also the temporal shape of the residual signal can be modeled. The ratio of residual signal power to input signal power is calculated and passed to encoder as a measure of “relevance” of the noise-like signal component.

7.A.2.2 HILN parameter encoder

The extracted parameters of the harmonic, individual line and noise parts of the signal are quantized and encoded to generate the bitstream output of the HILN encoder.

The allocation of the bits available in a frame to the parameters for the three parts of the signal is determined by the harmonic and noise component “relevance” measures calculated during the parameter estimation.

7.A.2.2.1 Harmonic Line Parameter Quantization

The number of bits available for the harmonic line parameters depends on the “relevance” measure of the harmonic signal component. If this measure is low, the number of harmonic lines encoded can be less than the number of lines extracted. This corresponds to a bandwidth limitation of the harmonic signal.

The fundamental frequency is quantized with 2048 steps on a logarithmic scale ranging from 20 Hz to 4 kHz. The “stretching” parameter is quantized with 35 steps on a uniform scale ranging from –17/16000 to +17/16000.

To describe the spectrum of the harmonic tone, the autocorrelation function of the harmonic signal is calculated. From this LAR LPC coefficients are derived that approximately model the spectral envelope of the harmonic signal. This process is similar to the LPC specral modeling used for the noise signal. Besides the LAR parameters also the power of the harmonic tone is calulated.

For a new harmonic tone, the indices of the quantised fundamental frequency and amplitude are directly written to the bitstream, while for a continued harmonic tone the index differences to the previous frame are coded with an entropy code.

7.A.2.2.2 Individual Line Parameter Quantization

In the *Quantization and Coding* unit, the parameters are processed in the order as they are obtained from the *Analysis/Synthesis Loop*, since this order corresponds to the relevancy with respect to the reproduction of the sound. This unit is able to generate two bitstreams, one *basic bitstream* which allows the generation of the basic quality audio signal, and an *enhancement bitstream* which can be used in applications where a difference signal between input and decoder output is needed, e.g. for scalability. The *basic bitstream* mainly contains the frequency and amplitude parameters, while the *enhancement bitstream* contains phase parameters and information for finer quantization of frequency and envelope parameters.

For each frame of the input signal, a number of bits according to the desired bitrate is transmitted. In each frame an *envelope bit* is transmitted, which indicates whether envelope parameters are used or not. If this bit is set, the 3 envelope parameters are transmitted, and for each transmitted line an additional *line envelope bit* is transmitted, which indicates, whether a constant amplitude or the envelope is to be used for the synthesis of the corresponding line.

Since the human auditory system is not very sensitive to phase changes, only the frequency and amplitude information of the spectral lines are coded and transmitted in the *basic bitstream* to obtain a signal with the basic audio quality. But in this case, it is necessary to provide information for the decoder which enables it to generate a signal free of phase discontinuities at frame boundaries. Therefore the first processing stage detects lines which continue from one frame to another. If a line is to be continued from the previous frame, only the frequency and amplitude changes are quantized and transmitted instead of the absolute frequency and amplitude values. For this purpose the frequency and amplitude parameters of the *i*-th line in the current frame *m* are compared with those of the *k*-th line in the previous frame *m*-1 for all possible combinations of *i* and *k*. The line continuation is used, if the relative frequency change

$$q_f(i,k) = \frac{\left| f_i(m) - f_k(m-1) \right|}{f_i(m)}$$

does not exceed a given threshold $q_{f,max}$ and if the ratio of amplitudes

$$q_a(i,k) = \begin{cases} a_i(m)/a_k(m-1) & \text{if } a_i(m) \geq a_k(m-1) \\ a_k(m-1)/a_i(m) & \text{if } a_i(m) < a_k(m-1) \end{cases}$$

lies within the interval [1...$q_{a,max}$]. If there is more than one possibility to continue a line from the previous frame, that line in the previous frame is selected, for which the following similarity criterion reaches its maximum:

$$Q = \frac{q_{f,max} - q_f(i,k)}{q_{f,max}} \cdot \frac{q_{a,\max} - q_a(i,k)}{(q_{a,\max} - 1)\, q_a(i,k)}$$

The frequencies and amplitudes of the individual lines are quantized according to a Bark-like frequency scale and a logarithmic amplitude scale. For each line of the previous frame, a *continuation bit* is transmitted in the bitstream, which indicates whether the line is continued in the current frame or not. For new lines, the indices for the quantised frequency and amplitude are encoded using a SubDivisionCode (SDC) as described below. For all lines continued from the previous frame, the frequency and amplitude index difference is encoded with an entropy code.

Since the transmission of absolute frequency and amplitude values requires more bits per line than for the relative values, the number of lines transmitted per frame is varied in if a constant bitrate for the *basic bitstream* is desired.

Since the *basic bitstream* does not contain phase information, it is not useful to calculate a residual error signal by subtracting the corresponding decoder output signal from the input signal. In order to enable scalability modes, in which the residual signal is transmitted in a bitstream of higher bitrate, an additional *enhancement bitstream* is generated. It is constructed in the following way:

- if the envelope parameters are transmitted in the *basic bitstream*, additional bits for finer quantization of the 3 envelope parameters are transmitted

- if a line is starting, i.e. not continued from the previous frame, and its frequency exceeds a given threshold, additional bits for finer quantization of the absolute frequency are transmitted

- for each line the phase parameter is transmitted after uniform quantization

The number of bits per frame in the *enhancement bitstream* can vary, this has to be taken into account in the calculation of available bits for the coding of the residual error.

Since the position of a continued line in the current frame depends on the position of its predecessor in the previous frame, a bit allocation algorithm is used which ensures that the *N* lines transmitted in the current frame are always the *N* most relevant lines found by the *Analysis/Synthesis Loop*.

The system delay of the encoder is 1.5 times the frame length. This delay results from the length of the frame itself plus an additional delay of (0.5) times the frame length caused by the shifted overlapping window used for the frequency estimation.

SDC-Encoding:

k : number of codewords (0...k-1)
i : value to encode
tab: table containing domain limits

```
void SDCEncode(int k,int i,int *tab)
{
    int  *pp;
    int  g,dp,min,max,cwl;
    long    cw;

    cw = cwl = 0;
    min = 0;
    max = k-1;
    pp = tab+16;
    dp = 16;

    while (min != max)
    {
        if (dp) g = (k*(*pp)) >> 10; else g = (max+min) >> 1;
        dp >>= 1;
        cw <<= 1;
        cwl++;
        if (i <= g) { pp -= dp; max = g; } else { cw |= 1; pp += dp; min = g+1; }
```

```
        }
        PutBits(cw,cwl);
    }
```

PutBits() writes the codeword to the bitstream, where the LSBs in cw are the “vlclbf” codeword and cwl determines number of bits to be transmitted.

7.A.2.2.3 Noise parameter quantization

The number of noise parameters that are quantized and encoded depends on the “relevance” measure of the noise signal component. If it is very low, no noise parameters are transmitted. For higher values of this measure an adequate number of LAR parameters are quantised and coded. Due to the properties of the reflection coefficients, the number of LAR parameters transmitted can be decided during bit-allocation in the encoder and no re-calculation of these parameters is required.

If the noiseEnvFlag is set then also the additional set of noise envelope parameters is quantized and coded.

7.A.2.3 HILN bitrate scalability

Due to the parametric signal representation utilized by the HILN parametric coder, it is well suited for applications requiring bitrate scalable coding. In such an application, the bitrate received by a decoder can be adapted dynamically to the properties of the transmission link or chosen according to some other rules. In case of a reduced-rate bitstream, only the parameters of the perceptually most relevant signal components (individual lines, harmonic tone, noise) are transmitted. In case of a full-rate bitstream also the parameters of additional signal components (e.g. individual lines) which are perceptually less relevant than those in the reduced-rate bitstream as well as additional parameters which refine the description of signal components already present in the reduced-rate bitstream are transmitted.

This bitrate scalability for HILN bitstreams can be implemented using the base and extension layer bitstreams as described in the normative part of the standard or by a dynamically controlled parameter encoding as described below.

7.A.2.3.1 HILN Bitrate Scalability by Dynamically Controlled Parameter Encoding

To implement bitrate scalability by means of dynamically controlled parameter encoding, it is utilized that both the HILN Parameter Extraction and the HILN Parameter Encoder can be operated independently. The parameters generated by the Parameter Extraction tool can be fed to multiple Parameter Encoder tools, each of which generates a bitstream with a different bitrate. This is computationally very advantageous since HILN encoder complexity is mainly determined by the Parameter Extraction tool. It is also possible to store the unquantized parameters generated by the Parameter Extraction tool in a file. Then the Parameter Encoder tool can be used to generate a bitstream with the currently requested bitrate from the parameters stored in this file.

7.A.3 Music/Speech Mixed Encoder tool

In MPEG-4 Audio the parametric coder is utilized for coding natural audio signals at very low bitrates ranging from 2 kbit/s to about 16 kbit/s. The parametric coder provides two sets of tools suited for coding speech and non-speech audio signals respectively:

- The Harmonic Vector Excitation (HVXC) tools are suited for coding speech signals at 2 kbit/s and 4 kbit/s.
- The Harmonic and Individual Lines plus Noise (HILN) tools are suited for coding non-speech audio signals at bitrates of about 4 kbit/s and above.

In the HVXC only or HILN only mode, the encoding mode is selected manually during encoding and the selected mode is used for all of the audio signal being encoded.

In this Subclause, an integrated parametric coder that utilizes the HVXC and HILN tools alternatively or simultaneously is described. This integrated coder automatically selects the coding tools that are suited best for the actual input signal characteristics. In case of a speech signal the HVXC tools are used and for music the HILN tools are used. This selection is done based on the decision of an automatic speech/music classification tool. For signals that are a mixture of speech and music it is also possible to use the HVXC and HILN tools simultaneously.

7.A.3.1 Music/Speech classification tool

This is a tool for the parametric speech coder coder, which enables automatic music/speech identification for the parametric speech/audio coder (HVXC and HILN). The tool makes decisions using internal parameters of the HVXC.

The features of input sequences used for the identification are: behavior of “pitch strength” and “frame energy”. In general, speech has higher “pitch strength” and frequent and higher energy change than music.

This music/speech classification tool can be applied in two ways:

- The first 5 seconds of the signal to be encoded are analyzed by the classification tool and then either HVXC or HILN is selected to encode the signal according to the speech/music decision.

- The classification tool is operated continuously and its current speech/music decision is used to select HVXC or HILN for the current frame. In this application the decision delay of 5 sec has to be taken into account.

7.A.3.1.1 Frame energy

Frame Energy P is computed as:

$$P = \sum_{n=0}^{159} s(n)^2$$

where s(n) is the input signal.

In this case, frames with energy levels higher than a pre-determined minimum level are used (ex. > -78 dB). A short-term average frame energy is defined as

$$Pav = \sum_{t=0}^{3} P\{t\} / 4$$

which is computed from the last four Frame energies.

A difference between frame energy and short term average frame energy is computed as:

$$Pd[frm] = |P - Pav| / Pav$$

Pd[frm] is kept for around 250 frames (5 seconds).

7.A.3.1.2 Pitch Strength

In HVXC, maximum autocorrelation of LPC residual (r0r) is computed during pitch detection process. r0r are kept for around 250 frames.

7.A.3.1.3 Music/Speech decision

Mean and variance of frame energies and r0rs are computed respectively as:

$$Pd(av) = \sum_{frm=0}^{249} Pd[\,frm] / 250$$

$$Pd(va) = \sqrt{\sum_{frm=0}^{249} \left(Pd[\,frm] - Pd(av)\right)^2} \; / 250$$

$$r0r(av) = \sum_{frm=0}^{249} r0r[\,frm] / 250$$

$$r0r(va) = \sqrt{\sum_{frm=0}^{249} \left(r0r[\,frm] - r0r(av)\right)^2} \; / 250$$

Speech data has higher variances than music data in the same range of mean value of r0r. The matrix is classified to three areas.

(1) speech $r0r(va) \geq 0.153\, r0r(av) + 0.113$

(2) unknown $0.07\, r0r(av) + 0.137 < r0r(va) < 0.153\, r0r(av) + 0.113$

(3) music $0.07\, r0r(av) + 0.137 \geq r0r(va)$

If mean and variance are included in the area (1), the data is classified as speech. If they are in the area (3), the data is classified as music.

If the mean and variance exist in the area (2), the mean and variance of (differential) frame energy Pd are used additionally. Speech data has larger means and variances of Pd than music data. Speech and music data is separated into the following two areas.

(1) speech $Pd(va) \geq -0.5 Pd(av) + 0.8$

(2) music $Pd(va) < -0.5 Pd(av) + 0.8$

Using the above two criteria, speech and music are separated.

7.A.3.2 Integrated parametric coder

The integrated parametric coder can operate in the following modes:

PARAmode	Description
0	HVXC only
1	HILN only
2	switched HVXC / HILN
3	mixed HVXC / HILN

PARAmodes 0 and 1 represent the fixed HVXC and HILN modes. PARAmode 2 permits automatic switching between HVXC and HILN depending on the current input signal type. In PARAmode 3 the HVXC and HILN coders can be used simultaneously and their output signals are added (mixed) in the decoder.

The integrated parametric coder uses a frame length of 40 ms and a sampling rate of 8 kHz and can operate at 2025 bit/s or any higher bitrate. Operation at 4 kbit/s or higher is suggested.

7.A.3.2.1 Integrated Parametric encoder

For the “HVXC only” and “HILN only” modes the parametric encoder is not modified. The “switched HVXC / HILN” and “mixed HVXC / HILN” modes are described below.

7.A.3.2.2 Switched HVXC/HILN mode

Because the speech/music classification tool is based on the HVXC encoder, the HVXC encoder is operated continuously for every frame. The bitstream frame generated by the HVXC encoder and the input audio signal are stored in two FIFO buffers to compensate for the 5 sec delay of the speech/music decision. If a frame is classified as “speech” then PARAswitchMode is set to 0 and the HVXC bitstream frame available at the bitstream FIFO output is transmitted. In case of a “music” decision, then PARAswitchMode is set to 1 and the output of the signal FIFO buffer is encoded by the HILN encoder and this HILN bitstream frame is transmitted. If HVXC is used for a frame, the HILN encoder is reset (prevNumLine = 0).

7.A.3.2.3 Mixed HVXC/HILN mode

To operate the parametric codec in “mixed HVXC / HILN” mode, speech and music components of the input signal have to be separated. If both components are already available separately (e.g. speech and background music) encoding is straightforward.

Contents for Subpart 8

Subpart 8: Technical description of parametric coding for high quality audio

8.1 Scope

This subpart of ISO/IEC 14496-3 describes the MPEG-4 audio parametric coding scheme for compression of high quality audio. The short name is SSC (SinuSoidal Coding). At bit-rates around 24 kbit/s stereo and at a sampling rate of 44.1 kHz, the SSC coding scheme offers a quality that is interesting for a number of applications.

SSC employs four different tools that together parameterize an audio signal. These tools consist of transient modelling, sinusoidal modelling, noise modelling and stereo image modelling. One of the distinctive features of SSC is that it provides decoder support for independent tempo and pitch scaling at hardly any additional complexity.

Transient tool

The transient tool captures the highly dynamic events of the audio input signal. These events are efficiently modelled by means of a limited number of sinusoids that are shaped by means of an envelope.

Sinusoidal tool

The sinusoidal tool captures the deterministic events of the audio input signal. The slowly varying nature of sinusoidal components for typical audio signals is exploited by linking sinusoids over consecutive frames. By means of differential coding, the frequency, amplitude and phase parameters can be efficiently represented.

Noise tool

The noise tool captures the stochastic or non-deterministic events of the audio input signal. In the decoder, a white noise generator is used as excitation. A temporal and spectral envelope is applied to control the temporal and spectral properties of the noise in the audio signal.

Parametric stereo coding tool

The parametric stereo coding tool is able to capture the stereo image of the audio input signal into a limited number of parameters, requiring only a small overhead ranging from a few kbit/s for medium quality, up to about 9 kbit/s for higher quality. Together with a monaural downmix of the stereo input signal generated by the parametric stereo coding tool, the parametric stereo decoding tool is able to regenerate the stereo signal. It is a generic tool that in principle can operate in combination with any monaural coder. In Annex 8.A of this document a normative description of the combination of HE-AAC with the parametric stereo coding tool is provided. SSC can also operate in dual mono mode. In that case the parametric stereo coding tool is not employed. The parametric stereo tool is intended for low bit-rates.

8.2 Terms and definitions

8.2.1
Frame
Basic unit that can be decoded on itself (file header information is required for general decoder settings).

8.2.2
Laguerre filter
Filter structure used in the noise analysis and synthesis.

8.2.3
Audio frame
Contains all data to decode an SSC-coded frame as a stand-alone unit (file header information is required for general decoder settings). For audio frames with refresh_sinusoids==%1 and refresh_noise==%1 the complete frame can always be reconstructed; otherwise it is possible in the case of random access that parts of the signal cannot be reconstructed (e.g. sinusoidal continuations, noise).

8.2.4
Sub-frame
Fine granularity within a frame.

8.2.5
f_s
The sampling frequency in Hertz.

8.2.6
Segment
An interval of samples that can be synthesized on the basis of the parameters that correspond to a sub-frame. The segment size is 2*S (see Table 8.16).

8.2.7
Window
A function that is used to weigh synthesized samples within a segment such that a valid synthesis is obtained.

8.2.8
LSF
Line Spectral Frequency.

8.2.9
Overlap and add
An additive method of combining overlapping intervals during signal synthesis.

8.2.10
Linking process
A method to keep track of sinusoidal components over time.

8.2.11
birth
The first component of a sinusoidal track.

8.2.12
Continuation
A sinusoidal track component that is not at the start or the end of a track.

8.2.13
Death
The last component of a sinusoidal track.

8.2.14
SMR
Signal-to-masking ratio.

8.2.15
Partial
Sinusoid of a limited duration.

8.2.16
IID
Inter-channel Intensity Differences.

8.2.17
IPD
Inter-channel Phase Differences.

8.2.18
OPD
Overall Phase Differences.

8.2.19
ICC
Inter-channel Coherence.

8.3 Symbols and abbreviations

8.3.1 Arithmetic operators

⌊x⌋ Round x towards minus infinity

⌈x⌉ Round x towards plus infinity.

mod Modulus operator: $\mathrm{mod}(x, y) = x - \left\lfloor \frac{x}{y} \right\rfloor y$. Defined only for positive values of *x* and *y*.

$\Gamma(\alpha)$ Gamma distribution function, defined as $\Gamma(\alpha) = \int_0^\infty e^{-t} \cdot t^{\alpha-1} dt$.

8.3.2 Relation operators

x?y:z If x is true then y else z.

8.3.3 Mnemonics

The following mnemonics are defined to describe the different data types used in the coded bit-stream.

uimsbf Unsigned integer, most significant bit first.

simsbf Signed integer, most significant bit first.

bslbf Bitstream left bit first.

8.3.4 Ranges

[0, 10] A number in the range of 0 up to and including 10.

[0, 10> A number in the range of 0 up to but excluding 10.

8.3.5 Number notation

%X Binary number representation (e.g. %01111100).

$X Hexadecimal number representation (e.g. $7C).

X Numbers with no prefix use decimal representation (e.g. 124).

8.3.6 Definitions

S Number of samples in a sub-frame (see Table 8.16).

L Number of samples in a segment; L = 2*S.

numQMFSlots Number of QMF subband samples per ps_data() element. For SSC, this parameter is fixed to 24.

8.4 Payloads for the audio object type SSC

8.4.1 Decoder configuration (SSCSpecificConfig)

Table 8.1 — Syntax of SSCSpecificConfig()

Syntax	Num. bits	Mnemonic
SSCSpecificConfig (channelConfiguration)		
{		
decoder_level	**2**	**uimsbf**
update_rate	**4**	**uimsbf**
synthesis_method	**2**	**uimsbf**
if (channelConfiguration != 1)		
{		
mode_ext	**2**	**uimsbf**
if ((channelConfiguration == 2) && (mode_ext == 1))		
{		
reserved	**2**	**uimsbf**
}		
}		
}		

8.4.2 SSC Bitstream Payload

Table 8.2 —Syntax of ssc_audio_frame()

Syntax	Num. bits	Mnemonic
ssc_audio_frame ()		
{		
ssc_audio_frame_header()		
ssc_audio_frame_data()		
}		

Table 8.3 — Syntax of ssc_audio_frame_header()

Syntax	Num. bits	Mnemonic
ssc_audio_frame_header ()		
{		
refresh_sinusoids	**1**	**uimsbf**
refresh_sinusoids_next_frame	**1**	**uimsbf**
refresh_noise	**1**	**uimsbf**
for (ch = 0; ch < nrof_channels; ch++)		
{		
s_nrof_continuations[0][ch]	Note 1	**uimsbf**
}		
n_nrof_den	**5**	**uimsbf**
n_nrof_lsf	Note 1	**uimsbf**
freq_granularity	**2**	**uimsbf**
amp_granularity	**2**	**uimsbf**
phase_jitter_present	**1**	**uimsbf**
if (phase_jitter_present == 1)		
{		
phase_jitter_percentage	**2**	**uimsbf**
phase_jitter_band	**2**	**uimsbf**
}		
}		
Note 1: See description of s_nrof_continuations and n_nrof_lsf in section 8.5.2.		

Table 8.4 — Syntax of ssc_audio_frame_data()

Syntax	Num. bits	Mnemonic
ssc_audio_frame_data()		
{		
for (sf = 0; sf < nrof_subframes; sf++)		
{		
for (ch = 0; ch < nrof_channels; ch++)		
{		
ssc_mono_subframe(sf,ch)		
if ((channelConfiguration == 2) && (mode_ext == 1)		
&& (mod(sf+1,4)==0))		
{		
ps_data()		
}		
}		
}		
}		

Table 8.5 — Syntax of ssc_mono_subframe()

Syntax	Num. bits	Mnemonic
ssc_mono_subframe (sf,ch)		
{		
subframe_transients(sf, ch)		
subframe_sinusoids(sf, ch)		
subframe_noise(sf, ch)		
}		

Table 8.6 — Syntax of subframe_transients()

Syntax	Num. bits	Mnemonic
subframe_transients (sf, ch)		
{		
t_transient_present[sf][ch]	**1**	**uimsbf**
if (t_transient_present[sf][ch] == 1)		
{		

t_loc[sf][ch]	Note 1	**uimsbf**
if (**t_type[sf][ch]**==1)	**2**	**uimsbf**
{		
t_b_par[sf][ch]	**3**	**uimsbf**
t_chi_par[sf][ch]	**3**	**uimsbf**
t_nrof_sin[sf][ch]	**3**	**uimsbf**
t_nrof_sin[sf][ch]++		
for (i = 0; i < t_nrof_sin[sf][ch]; i++)		
{		
t_freq[sf][ch][i]	**9**	**uimsbf**
t_amp[sf][ch][i]	**5**	**uimsbf**
t_phi[sf][ch][i]	**5**	**simsbf**
}		
}		
}		
}		
Note 1: See description of t_loc in section 8.5.2.		

Table 8.7 — Syntax of subframe_sinusoids()

Syntax	Num. bits	Mnemonic
subframe_sinusoids(sf, ch)		
{		
n = 0; p=0; q=0;		
/ Continuations */*		
if (sf > 0)		
{		
noc = 0;		
while (tmp_cont[ch][noc] > 0) { noc++;}		
s_nrof_continuations[sf][ch] = noc;		
}		
if ((refresh_sinusoids == 1) && (sf == 0))		
{		
for (i = 0; i < s_nrof_continuations[sf][ch]; i++, n++)		
{		
s_cont[sf][ch][n] = ssc_huff_dec(huff_scont,**bs_codeword**);	**2..5**	**bslbf**
s_freq_coarse[sf][ch][n] = ssc_huff_dec(huff_sfreqc,**bs_codeword**);	**7..25**	**bslbf**
s_freq_fine[sf][ch][n]	**0..3**	**simsbf**
s_amp_coarse[sf][ch][n] = ssc_huff_dec(huff_sampca,**bs_codeword**);	**3..16**	**bslbf**
s_amp_fine[sf][ch][n]	**0..3**	**simsbf**
s_phi[sf][ch][n]	**5**	**simsbf**
if (s_cont[sf][ch][n] > 0)		
{		
s_adpcm_grid[sf][ch][n] = ssc_huff_dec(huff_sgrid,**bs_codeword**);	**3..7**	**bslbf**
s_delta_cont_freq_pha[sf+1][ch][p]	**2**	**uimsbf**
p++;		
}		
if (s_cont[sf][ch][n] > 1)		
{		
s_delta_cont_freq_pha[sf+2][ch][q]	**2**	**uimsbf**
q++;		
}		
}		
}		
else {		
for (i = 0; i < s_nrof_continuations[sf][ch]; i++, n++)		
{		
if (sf == 0)		
{		
s_cont[sf][ch][n] = ssc_huff_dec(huff_scont,**bs_codeword**);	**2..5**	**bslbf**
}		

else {		
s_cont[sf][ch][n] = tmp_cont[ch][n] - 1;		
}		
if (s_cont[sf][ch][n] > 0)		
{		
p++;		
}		
if (s_cont[sf][ch][n] > 1)		
{		
if ((refresh_sinusoids_next_frame == 0) || (nrof_subframes-sf > 2))		
{		
s_delta_cont_freq_pha[sf+2][ch][q]	**2**	**uimsbf**
}		
q++;		
}		
s_delta_cont_amp[sf][ch][n] =	**1..15**	**bslbf**
ssc_huff_dec(huff_sampcr[amp_granularity]**,bs_codeword**);		
}		
}		
/ Births */*		
s_nrof_births[sf][ch] = ssc_huff_dec(huff_nrofbirths**,bs_codeword**);	**3..15**	**bslbf**
if (s_nrof_births[sf][ch] > 0)		
{		
s_cont[sf][ch][n] = ssc_huff_dec(huff_scont**,bs_codeword**);	**2..5**	**bslbf**
s_freq_coarse[sf][ch][n] = ssc_huff_dec(huff_sfreqba**,bs_codeword**);	**7..21**	**bslbf**
s_freq_fine[sf][ch][n]	**0..3**	**simsbf**
s_amp_coarse[sf][ch][n] = ssc_huff_dec(huff_sampba**,bs_codeword**);	**3..15**	**bslbf**
s_amp_fine[sf][ch][n]	**0..3**	**simsbf**
s_phi[sf][ch][n]	**5**	**simsbf**
if (s_cont[sf][ch][n] > 0)		
{		
if ((refresh_sinusoids_next_frame == 0) || (nrof_subframes-sf > 1))		
{		
s_delta_cont_freq_pha[sf+1][ch][p]	**2**	**uimsbf**
}		
p++;		
}		
if (s_cont[sf][ch][n] > 1)		
{		
if ((refresh_sinusoids_next_frame == 0) || (nrof_subframes-sf > 2))		
{		
s_delta_cont_freq_pha[sf+2][ch][q]	**2**	**uimsbf**
}		
q++;		
}		
n++;		
for (i = 1; i < s_nrof_births[sf][ch]; i++, n++)		
{		
s_cont[sf][ch][n] = ssc_huff_dec(huff_scont**,bs_codeword**);	**2..5**	**bslbf**
s_delta_birth_freq_coarse[sf][ch][n] =	**5..23**	**bslbf**
ssc_huff_dec(huff_sfreqbr**,bs_codeword**);		
s_delta_birth_freq_fine[sf][ch][n]	**0..3**	**simsbf**
s_delta_birth_amp_coarse[sf][ch][n] =	**2..21**	**bslbf**
ssc_huff_dec(huff_sampbr**,bs_codeword**);		
s_delta_birth_amp_fine[sf][ch][n]	**0..3**	**simsbf**
s_phi[sf][ch][n]	**5**	**simsbf**
if (s_cont[sf][ch][n] > 0)		
{		
if ((refresh_sinusoids_next_frame == 0) || (nrof_subframes-sf > 1))		
{		

s_delta_cont_freq_pha[sf+1][ch][p]	**2**	**uimsbf**
}		
p++;		
}		
if (s_cont[sf][ch][n] > 1)		
{		
if ((refresh_sinusoids_next_frame == 0) || (nrof_subframes-sf > 2))		
{		
s_delta_cont_freq_pha[sf+2][ch][q]	**2**	**uimsbf**
}		
q++;		
}		
}		
}		
/* Keep track of sinusoids that continue in next sub-frame(s) */		
for (i = 0, k = 0; i < n; i++)		
{		
if (s_cont[sf][ch][i] > 0)		
{		
tmp_cont[ch][k] = s_cont[sf][ch][i];		
k++;		
}		
}		
}		
Note: The variables p, q are used as position indices for subframe+1 and subframe+2 respectively.		

Table 8.8 — Syntax of subframe_noise()

Syntax	Num. bits	Mnemonic
subframe_noise (sf, ch)		
{		
if ((refresh_noise == 1) && (sf == 0))		
{		
n_laguerre[ch]	**2**	**uimsbf**
n_laguerre_granularity[sf][ch]	**1**	**uimsbf**
for (i = 0; i < n_nrof_den; I++)		
{		
n_lar_den_coarse[sf][ch][i] = ssc_huff_dec(huff_nlag**,bs_codeword**);	**1..18**	**bslbf**
If (n_laguerre_granularity[sf][ch]==1)		
{		
n_lar_den_fine[sf][ch][i]	**2**	**simsbf**
}		
}		
n_gain[sf][ch]	**7**	**uimsbf**
n_lsf[sf][ch][0] = ssc_huff_dec(huff_nlsf**,bs_codeword**);	**2..9**	**bslbf**
for (i = 1; i < n_nrof_lsf; i++)		
{		
n_delta_lsf[sf][ch][i] = ssc_huff_dec(huff_nlsf**,bs_codeword**);	**2..9**	**bslbf**
}		
}		
else {		
if (mod(sf,2) == 0)		
{		
n_laguerre_granularity[sf][ch]	**1**	**uimsbf**
for (i = 0; i < n_nrof_den; i++)		
{		
n_delta_lar_den_coarse[sf][ch][i] = ssc_huff_dec(huff_nlag**,bs_codeword**);	**1..18**	**bslbf**
if(n_laguerre_granularity[sf][ch]==1)		

{
n_delta_lar_den_fine[sf][ch][i]
}
	2	**simsbf**

}
}
if (mod(sf,4) == 0)
{		

n_delta_gain[sf][ch] = ssc_huff_dec(huff_ngain**,bs_codeword**);	**1..12**	**bslbf**
if (**n_overlap_lsf** == 1)	**1**	**uimsbf**

{
for (i = n_nrof_overlap_lsf; i < n_nrof_lsf; i++)
{		

n_delta_lsf[sf][ch][i] = ssc_huff_dec(huff_nlsf**,bs_codeword**);	**2..9**	**bslbf**

}
}
else {		

n_lsf[sf][ch][0] = ssc_huff_dec(huff_nlsf**,bs_codeword**); for (i = 1; i < n_nr_of_lsf; i++)	**2..9**	**bslbf**

{
n_delta_lsf[sf][ch][i] = ssc_huff_dec(huff_nlsf**,bs_codeword**);
}
	2..9	**bslbf**

} } }		

Table 8.9 — Syntax of ps_data()

Syntax	Num. bits	Mnemonic
ps_data() {		
if (**enable_ps_header**) {	**1**	**uimsbf**
if (**enable_iid**) {	**1**	**uimsbf**
iid_mode	**3**	**uimsbf**
nr_iid_par = nr_iid_par_tab[iid_mode] nr_ipdopd_par = nr_ipdopd_par_tab[iid_mode]		
}		
if (**enable_icc**) {	**1**	**uimsbf**
icc_mode	**3**	**uimsbf**
nr_icc_par = nr_icc_par_tab[icc_mode]		
}		
enable_ext	**1**	**uimsbf**
}		
frame_class	**1**	**uimsbf**
num_env_idx	**2**	**uimsbf**
num_env = num_env_tab[frame_class][num_env_idx]		
if (frame_class) {		
for (e=0 ; e<num_env ; e++) {		
border_position[e]	**5**	**uimsbf**
}		
}		
for (e=0 ; e<num_env ; e++) {		
if (enable_iid) {		
iid_dt[e]	**1**	**uimsbf**
iid_data()		
}		
}		

for (e=0 ; e<num_env ; e++) {		
if (enable_icc) {		
icc_dt[e]	**1**	**uimsbf**
icc_data()		
}		
}		
if (enable_ext) {		
cnt = **ps_extension_size**	**4**	**uimsbf**
if (cnt == 15)		
cnt += **esc_count**	**8**	**uimsbf**
num_bits_left = 8 * cnt		
while (num_bits_left > 7) {		
ps_extension_id	**2**	**uimsbf**
num_bits_left -= 2		
ps_extension(ps_extension_id, num_bits_left)		
}		
fill_bits	num_bits_left	
}		
}		

Table 8.10 — Syntax of ps_extension()

Syntax	Num. bits	Mnemonic
ps_extension(ps_extension_id, num_bits_left)		
{		
if (ps_extension_id == 0) {		
if (**enable_ipdopd**) {	**1**	**uimsbf**
for (e=0 ; e<num_env ; e++) {		
ipd_dt[e]	**1**	**uimsbf**
ipd_data()		
opd_dt[e]	**1**	**uimsbf**
opd_data()		
num_bits_left -= ipd_bits + opd_bits + 2		Note 1
}		
}		
reserved_ps	**1**	**uimsbf**
num_bits_left -= 2		
}		
}		

Table 8.11 — Syntax of iid_data()

Syntax	Num. bits	Mnemonic
iid_data()		
{		
if (iid_dt[e]) {		
for (b=0 ; b<nr_iid_par; b++) {		
iid_par_dt[e][b] = ssc_huff_dec(huff_iid_dt[iid_quant]**,bs_codeword**);	**1…20**	Note 2
}		
}		
else {		
for (b=0 ; b<nr_iid_par; b++) {		
iid_par_df[e][b] = ssc_huff_dec(huff_iid_df[iid_quant]**,bs_codeword**);	**1…18**	Note 2
}		
}		
}		

Table 8.12 — Syntax of icc_data()

Syntax	Num. bits	Mnemonic

icc_data()
{
if (icc_dt[e]) {
for (b=0 ; b<nr_icc_par; b++) {
icc_par_dt[e][b] = ssc_huff_dec(huff_icc_dt**,bs_codeword**);
}
}
else {
for (b=0 ; b<nr_icc_par; b++) {
icc_par_df[e][b] = ssc_huff_dec(huff_icc_df**,bs_codeword**);
}
}
}
	
	1…14
	
	1…13
		
		bslbf
		
		bslbf

Table 8.13 — Syntax of ipd_data()

Syntax	Num. bits	Mnemonic

ipd_data()
{
if (ipd_dt[e]) {
for (b=0 ; b<nr_ipdopd_par; b++) {
ipd_par_dt[e][b] = ssc_huff_dec(huff_ipd_dt**,bs_codeword**);
}
}
else {
for (b=0 ; b<nr_ipdopd_par; b++) {
ipd_par_df[e][b] = ssc_huff_dec(huff_ipd_df**,bs_codeword**);
}
}
}
	
	1…5
	
	1…4
		
		bslbf
		
		bslbf

Table 14 — Syntax of opd_data()

Syntax	Num. bits	Mnemonic

opd_data()
{
if (opd_dt[e]) {
for (b=0 ; b<nr_ipdopd_par; b++) {
opd_par_dt[e][b] = ssc_huff_dec(huff_opd_dt**,bs_codeword**);
}
}
else {
for (b=0 ; b<nr_ipdopd_par; b++) {
opd_par_df[e][b] = ssc_huff_dec(huff_opd_df**,bs_codeword**);
}
}
}
	
	1…5
	
	1…5
		
		bslbf
		
		bslbf

Note 1: ipd_bits and opd_bits represent the number of bits read by ipd_data() and opd_data() respectively. Note 2: the index iid_quant into huff_iid_df is obtained from Table 8.24.		

8.5 Semantics

8.5.1 SSCSpecificConfig

channelConfiguration – Channel configuration as defined in ISO/IEC-14496-3:edition 2001 subpart 1, paragraph 1.6.3.4.

decoder_level – Complexity bounds for decoder settings. A decoder that supports a certain level of complexity is not able to decode a bit-stream that is encoded according a higher level of complexity. This decoder is however able to decode a bit-stream that is encoded according to a lower level of complexity (see Table 8.15).

Table 8.15 — Decoder level

decoder_ level	Level of complexity	max_nrof_sinusoids	max_nrof_den	#bits for s_nrof_ continuations	#bits for n_nrof_lsf
00	Reserved	Na	Na	Na	Na
01	Medium	60	24	6	4
10	Reserved	Na	Na	Na	Na
11	Reserved	Na	Na	Na	Na

max_nrof_sinusoids - Maximum number of sinusoids that is allowed (sinusoids under Meixner transients not included).

max_nrof_den - Maximum value for n_nrof_den.

update_rate – Four bits indicating the sub-frame size S. Table 8.16 shows the relationship between update_rate and the sub-frame size S in samples.

Table 8.16 — Update rate

update_rate	S	update_rate	S
0000	Reserved	1000	Reserved
0001	Reserved	1001	Reserved
0010	Reserved	1010	Reserved
0011	Reserved	1011	Reserved
0100	384	1100	Reserved
0101	Reserved	1101	Reserved
0110	Reserved	1110	Reserved
0111	Reserved	1111	Reserved

synthesis_method – Two bits providing information on the preferred synthesis for the specific encoded program (see Table 8.17).

Table 8.17 — Synthesis method

Synthesis_method	Optimal synthesis
00	Overlap and Add
01	Reserved
10	Reserved
11	Reserved

mode_ext – In combination with channelConfiguration the mode_ext bits provide the full channel configuration. The number of bits is dependent on the channelConfiguration (see Table 8.18).

Table 8.18 — Channel configuration

channelConfiguration	# bits for mode_ext	nrof_channels
1	0	1
2	2	According to mode_ext
0, 3 … 15	Na	Na

For channelConfiguration == 2, Table 8.19 applies:

Table 8.19 – Channel configuration in case channelConfiguration == 2

mode_ext	Full channel configuration
00	Dual mono (ch0=left, ch1=right)
01	Parametric Stereo
10	Reserved
11	Reserved

reserved – Two reserved bits; should be set to %0.

8.5.2 Decoding of SSC Bitstream Payload

ssc_audio_frame() – syntactic element that contains a single SSC frame

ssc_audio_frame_header() – syntactic element that contains the header data for a single SSC frame

ssc_audio_frame_data() – syntactic element that contains the data for a single SSC frame

ssc_huff_dec() – Huffman decoding procedure. See Annex 8.B.

refresh_sinusoids – One bit indicating how sinusoidal continuations of the first sub-frame in a frame are encoded. If this bit equals %0, the continued track data is differentially coded with respect to the last sub-frame of the previous frame. If this bit equals %1, the continued track data in the first sub-frame of a frame are coded as absolute values.

refresh_sinusoids_next_frame – One bit providing an additional frame look ahead for the ADPCM decoding of sinusoidal parameters. If this bit is set to %1, the next frame is a refresh frame. In that case the bit refresh_sinusoids shall be set to %1 in the next frame. If this bit is set to %0, the next frame is not a refresh frame.

refresh_noise – One bit indicating how noise parameters of the first sub-frame in a frame are encoded. If this bit equals %0, the noise parameters are differentially coded with respect to the last sub-frame of the previous frame. If this bit equals %1, the noise parameters in the first sub-frame of a frame are coded as absolute values.

s_nrof_continuations[sf][ch] – For sub-frame sf and channel ch, this value represents the number of continuations. In the case sf==0 the value of s_nrof_continuations is provided in the bit-stream. For the remaining values of sf, the value of s_nrof_continuations is obtained implicitly as described in section 8.6.2.1. The number of bits required for s_nrof_continuations[0][ch] depends on the maximum number of allowed sinusoids, which is dependent of the decoder complexity, indicated by decoder_level. This relation is shown in Table 8.15.

n_nrof_den – Number of denominator LAR coefficients of the FIR filter for noise generation.

n_nrof_lsf - Number of LSF coefficients used for the generation of envelope for noise generation. The number of bits required for n_nrof_lsf depends on the decoder complexity, indicated by decoder_level. This relation is shown in Table 8.15.

freq_granularity – The granularity of the differentially or absolute coded frequency parameters used in subframe_sinusoids(). This parameter determines the number of bits to be read for the fine part of the frequency parameters.

amp_granularity – The granularity of the differentially or absolute coded amplitude parameters used in subframe_sinusoids(). This parameter determines the Huffman table to be used or the number of bits to be read for the fine part of the amplitude parameters.

phase_jitter_present – One bit to indicate presence of phase jitter parameters. If this bit equals %0, no phase jitter is present. If this bit equals %1, phase jitter is present.

phase_jitter_percentage – This is a two bit unsigned integer identifying a distance percentage. The full distance equals half a quantisation step. The maximum jitter applied to the frequency components is

$$max\_jitter = 2^{\text{freq\_granularity}-1}\,\frac{\text{phase\_jitter\_percentage}+1}{2^2}.$$

phase_jitter_band – Two bits identifying the frequency representation level from which phase jitter must be applied. Table 8.20 provides the relation between phase_jitter_band and $f_{jitter,min}$.

Table 8.20 — Phase jitter band expressed in representation levels

phase_jitter_band	frequency representation level $f_{jitter,min}$
00	0
01	800
10	1600
11	2400

nrof_subframes – the number of sub-frames in one frame. This value is fixed to 8.

ssc_mono_subframe() – syntactic element that contains the data for a single SSC sub frame.

ps_data() – syntactic element that contains the parametric stereo data.

subframe_transients() – syntactic element that contains the transient data for a single SSC sub frame.

subframe_sinusoids() – syntactic element that contains the sinusoid data for a single SSC sub frame.

subframe_noise() – syntactic element that contains the noise data for a single SSC sub frame.

t_transient_present[sf][ch] – One bit indicating if a transient is present in sub-frame sf, channel ch. If t_transient_present[sf][ch]==%1, a transient is present. If t_transient_present[sf][ch]==%0, no transient is present.

t_loc[sf][ch] – Indication of the location of the transient in sub-frame sf of channel ch, indicated in the number of samples from the start of the sub-frame. The valid range for t_loc is [0,S>. The number of bits that is used to represent t_loc is calculated according to

$$\lceil \log_2(S) \rceil,$$

where S represents the sub-frame size in samples.

t_type[sf][ch] – Two bits to indicate the transient type of the transient in sub-frame sf of channel ch (see Table 8.21).

Table 8.21 — Transient types

t_type	Type
00	Step
01	Meixner
10	Reserved
11	Reserved

t_b_par[sf][ch] – For a transient of the Meixner type in sub-frame sf of channel ch, these 3 bits hold the value for the attack of the transient envelope, denoted as the '*b*-parameter'. Allowed values for t_b_par are [0, 1, 2, 3]. The remaining values are reserved. The value *b* is calculated as

b = t_b_par + 2.

t_chi_par[sf][ch] – For a transient of the Meixner type in sub-frame sf of channel ch, these 3 bits hold the value for the decay of the transient envelope, denoted as the 'ξ-parameter'. Allowed values for t_chi_par are [0, 1, 2, 3]. The remaining values are reserved. The value ξ is tabulated in Table 8.22.

Table 8.22 — Quantised values for b and ξ

ξ		t_b_par			
		0	1	2	3
t_chi_par	0	0.9688	0.9685	0.9683	0.9681
	1	0.9763	0.9756	0.9750	0.9744
	2	0.9839	0.9827	0.9817	0.9807
	3	0.9914	0.9898	0.9884	0.9870

t_nrof_sin[sf][ch] – For a transient of the Meixner type in sub-frame sf of channel ch, these 3 bits represent the number of sinusoids that are present under the envelope. The number of sinusoids under the Meixner envelope is equal to the value in the stream plus one.

t_freq[sf][ch][i] – For a transient of the Meixner type in sub-frame sf of channel ch, these bits represent the frequency in radians of the i-th sinusoid under the transient envelope.

$$tf_q[i] = \frac{2\pi}{f_s}\frac{10^{\frac{\text{t\_freq[sf][ch][i]}}{11.4\cdot 21.4}}-1}{0.00437},$$

where tf_q represents the dequantized absolute frequency in radians.

t_amp[sf][ch][i] – For a transient of the Meixner type in sub-frame sf of channel ch, these bits represent the amplitude of the i-th sinusoid under the transient envelope.

$$ta_q[i] = ta_b^{2\cdot \text{t\_amp[sf][ch][i]}},$$

where ta_b represents the log quantisation base (maximum error=1.5dB), $ta_b = 1.1885$. ta_q represents the dequantized absolute amplitude.

t_phi[sf][ch][i] – For a transient of the Meixner type in sub-frame sf of channel ch, these bits represent the phase of the i-th sinusoid under the transient envelope. The decoded value is converted into a phase value in radians in the range [-π, π> and is specified for the start of the transient.

$$tp_q[i] = 2\cdot tp_e \cdot \text{t\_phi[sf][ch][i]},$$

where tp_e represents the absolute phase error ($tp_e = \frac{\pi}{32}$) and tp_q represents the dequantized absolute phase (in radians). The allowed range for t_phi is [-16, 15].

noc – Local variable that counts the number of continuations in the previous sub-frame.

tmp_cont[ch][noc] – Local array that contains a copy of the s_cont-parameters of the previous sub-frame, needed for parsing the stream correctly (extract number of continuations and keep track of how many sub-frames the sinusoidal track must be continued in the current frame).

s_cont[sf][ch][n] – For sub-frame sf and channel ch, this value indicates how many sub-frames component n will be continued in the current frame (if the component continues also in the next frame, one must be added to the number of sub-frames it continues in the current frame. If the value is 0 this indicates component n stops at sub-frame sf, which is called a death). The valid range for s_cont is [0, 9].

s_freq_coarse[sf][ch][n] – For sub-frame sf and channel ch, this value represents the coarse frequency parameter of the n-th sinusoid.

s_freq_fine[sf][ch][n] – For sub-frame sf and channel ch, this signed integer represents a higher level of detail to the coarse frequency parameter. The number of bits to be read amounts to (3 – freq_granularity). The frequency representation level f_{rl} is the sum of coarse frequency, fine frequency scaled to the granularity grid.

$$f_{rl}[n] = \text{s\_freq\_coarse}[sf][ch][n] + \text{s\_freq\_fine}[sf][ch][n] \cdot 2^{\text{freq\_granularity}}.$$

Phase jitter is only applied in combination with tempo and pitch scaling. If phase_jitter_present == %1 and $f_{rl} > f_{jitter,min}$, the phase jitter parameter is

$$f_{jitter} = \lfloor max\_jitter \cdot (2x - 1) + 0.5 \rfloor,$$

where x holds a random number, uniformly distributed between 0 and 1, generated for each frequency parameter in the sub-frame, matching the requirement above. The decoded value is converted into a dequantized absolute frequency value f_q in radians, using the following equation:

$$f_q[n] = \frac{2\pi}{f_s} \frac{10^{\frac{f_{rl}[n]}{91.2 \cdot 21.4}} - 1}{0.00437}.$$

s_amp_coarse[sf][ch][n] – For sub-frame sf and channel ch, this value represents the coarse amplitude parameter of the n-th sinusoid.

s_amp_fine[sf][ch][n] – For sub-frame sf and channel ch, this parameter represents a higher level of detail to the coarse amplitude parameter. The number of bits to be read amounts to (3 – amp_granularity). The amplitude representation level sa_{rl} is the sum of coarse amplitude, fine amplitude scaled to the granularity grid

$$sa_{rl}[n] = \text{s\_amp\_coarse}[sf][ch][n] + \text{s\_amp\_fine}[sf][ch][n] \cdot 2^{\text{amp\_granularity}}.$$

The decoded value is converted into a dequantized linear amplitude value sa_q in the range [1, 2^{15}-1] conforming to

$$sa_q[n] = sa_b^{2 \cdot sa_{rl}[n]}.$$

Where $sa_b = 1.0218$ is the log quantization base. Its value corresponds to a maximum error of 0.1875 dB.

s_phi[sf][ch][n] – For sub-frame sf and channel ch, this represents the phase parameter of the n-th sinusoid. This value is converted into a phase value in radians in the range [-π, π> conforming to

$$sp_q[n] = 2 \cdot sp_e \cdot \text{s\_phi[sf][ch][n]},$$

where sp_e represents the absolute phase error ($sp_e = \frac{\pi}{32}$) and sp_q represents the dequantized absolute phase (in radians) The allowed range for s_phi is [-16, 15]; the representation level +16 is represented by –16 (because +π == -π).

s_adpcm_grid[sf][ch][n] – For sub-frame sf and channel ch, this value represents the initial index into Table 8.35 as used in the ADPCM decoder for the n-th sinusoid. This table is used to decode the sinusoidal information.

s_delta_cont_freq_pha[sf][ch][n] – For sub-frame sf and channel ch, this value represent the representation levels for the n-th sinusoid, that serve as input to the ADPCM decoder. In order to compensate for the 2 sub-frames delay in the decoder, the representation levels are transmitted 2 sub-frames in advance. In the bit-stream syntax, future representation levels are indicated by indices sf+1 and sf+2, indicating the representation levels of the two sub-sequent sub-frames respectively. In the case sf+1 or sf+2 exceeds nrof_subframes, then the representation level is assigned to the next frame. In this case, the new number of the sub-frame in the next frame is (sf+1) - nrof_subframes or (sf+2) - nrof_subframes respectively.

s_delta_cont_amp[sf][ch][n] – For sub-frame sf and channel ch, this represents the differential amplitude parameter of the n-th sinusoid. This value is converted into a linear amplitude value in the range [1, 2^{15}-1] conforming to

$$sa_{rl}[n] = sa_{rl,psf} + \text{s\_delta\_cont\_amp[sf][ch][n]},$$

where sa_{rl} represents the amplitude representation level and $sa_{rl,psf}$ represents the amplitude representation level in the previous sub-frame. For dequantization of sa_{rl} into sa_q see s_amp_fine. In the case the amplitude granularity, amp_granularity of the current frame is different from that of the previous frame, prior to applying the differential encoded values, the granularity of the value of the previous frame is converted to the granularity of the current frame according to

$$sa_{rl,psf} = 2^{\text{amp\_granularity}} \left\lfloor \frac{sa'_{rl,psf}}{2^{\text{amp\_granularity}}} + 0.5 \right\rfloor,$$

where $sa'_{rl,psf}$ represents the amplitude representation level of the previous sub-frame and amp_granularity represents the granularity of the current sub-frame.

s_nrof_births[sf][ch] – For sub-frame sf and channel ch, this value represents the number of births. The allowed range is [0, max_nrof_sinusoids-s_nrof_continuations[sf][ch]].

s_delta_birth_freq_coarse[sf][ch][n] – For sub-frame sf and channel ch, this value represents the differential, coarse frequency parameter of the n-th sinusoid.

s_delta_birth_freq_fine[sf][ch][n] – For sub-frame sf and channel ch, this represents a higher level of detail to the coarse differential frequency parameter. The number of bits to be read is (3 – freq_granularity). The delta frequency representation level df_{rl} is

$$df_{rl}[n] = \text{s\_delta\_birth\_freq\_coarse[sf][ch][n]} + \text{s\_delta\_birth\_freq\_fine[sf][ch][n]} \cdot 2^{\text{freq\_granularity}}$$

The decoded value of the n-th sinusoid is converted into a frequency value in Hertz, using the frequency representation level of the previous birth f_{rl} of sub-frame sf (the (n-1)th sinusoid):

$$f_{rl}[n] = f_{rl}[n-1] + df_{rl}[n],$$

where f_{rl} represents the frequency representation level. Modification of f_{rl} due to phase jitter uses the same rules as stated under s_freq_fine. For dequantization of f_{rl} into f_q see also s_freq_fine.

s_delta_birth_amp_coarse[sf][ch][n] – For sub-frame sf and channel ch, this represents the differential, coarse amplitude parameter of the n-th sinusoid.

s_delta_birth_amp_fine[sf][ch][n] – For sub-frame sf and channel ch, this represents a higher level of detail to the coarse amplitude parameter. The number of bits to be read is (3 – amp_granularity). The delta amplitude representation level sda_{rl} is

$$sda_{rl}[n] = \text{s\_delta\_birth\_amp\_coarse[sf][ch][n]} + \text{s\_delta\_birth\_amp\_fine[sf][ch][n]} \cdot 2^{\text{amp\_granularity}}$$

The decoded value for the n-th sinusoid is converted into a linear amplitude value, using the amplitude representation level of the previous birth sa_{rl} (the (n-1)th sinusoid):

$$sa_{rl}[n] = sa_{rl}[n-1] + sda_{rl}[n],$$

where sa_{rl} represents the amplitude representation level. For dequantization of sa_{rl} into sa_q see s_amp_fine.

n_laguerre[ch] – λ coefficient of the Laguerre filter for noise synthesis, see Table 8.23.

Table 8.23 — Possible values for λ

n_laguerre	λ
00	0
01	0.5
10	0.7
11	Reserved

n_laguerre_granularity[sf][ch] – 1 bit denoting quantisation accuracy of the Laguerre coefficients.

n_lar_den_coarse[sf][ch][i] – for sub-frame sf and channel ch this represents the denominator LAR coefficient number i.

n_lar_den_fine[sf][ch][i] – for sub-frame sf and channel ch this represents a higher level of detail to the coarse denominator LAR coefficient parameter. The representation level $nlar_{rl}$ is the sum of coarse denominator LAR and fine denominator LAR:

$nlar_{rl}[i]$ = n_lar_den_coarse[sf][ch][i] + n_lar_den_fine[sf][ch][i]

and is converted to a LAR coefficient according to:

$nlar_q[i] = nlar_{rl}[i] * \Delta_{LAR}$.

For the definition of Δ_{LAR} see section 8.6.3.3.1. Section 8.6.3.3.2 and 8.6.3.3.3 show how to convert LARs to reflection- or a-parameters.

n_gain[sf][ch] – For sub-frame sf and channel ch this value represents the gain coefficient. The gain representation level $ngain_{rl}$ is obtained as:

$ngain_{rl}$ = n_gain[sf][ch]

n_lsf[sf][ch][i] – For sub-frame sf and channel ch this value represents the LSF coefficient number i. The allowed range for n_lsf is [0,255]. The dequantised LSF parameters $nlsf_q$ are obtained as

$nlsf_q[i]$ = n_lsf[sf][ch][i] * π / 256 + π / 512

n_delta_lsf[sf][ch][i] – For sub-frame sf and channel ch this value represents the differential LSF coefficient number i. They are obtained using the following algorithm

for $i > 0$:

$nlsf_q[i] = (nlsf_q[i\text{-}1] + \text{n\_delta\_lsf[sf][ch][i]}) * \pi / 256 + \pi / 512$.

n_delta_lar_den_coarse[sf][ch][i] – for sub-frame sf and channel ch this represents the differential, denominator LAR coefficient number i.

n_delta_lar_den_fine[sf][ch][i] – for sub-frame sf and channel ch this represents a higher level of detail to the coarse denominator LAR coefficient parameter. The representation level $ndlar_{rl}$ is the sum of the differential coarse denominator LAR and differential fine denominator LAR:

$ndlar_{rl}[i]$ = n_delta_lar_den_coarse[sf][ch][i] + n_delta_lar_den_fine[sf][ch][i]

and is converted to a LAR coefficient according to:

$nlar_{rl}[i] = nlar_{rl,psf}[i] + ndlar_{rl}[i]$.

where $nlar_{rl}[i]$ and $nlar_{rl,psf}[i]$ represent the LAR representation level of the current and previous sub-frame respectively. Note that in the case the n_laguerre_granularity changes from %1 to %0 going from sub-frame sf-1 to sf, the value $nlar_{rl,psf}[i]$ is first converted to the coarsest quantisation grid according to:

$nlar_{rl,psf}[i] = 4 * \lfloor nlar'_{rl,psf}[i] / 4 + 0.5 \rfloor$.

where $nlar'_{rl,psf}[i]$ represents the LAR representation level of the previous sub-frame. For further processing of $nlar_{rl}$ into reflection or a-parameters see sections 8.6.3.3.2 and 8.6.3.3.3.

n_delta_gain[sf][ch] – For sub-frame sf and channel ch this value represents the differential gain coefficient, and is converted to a representation level $ngain_{rl}$ according to:

$ngain_{rl} = ngain_{rl,p4sf}$ + n_delta_gain[sf][ch],

where $ngain_{rl,p4sf}$ represents the gain representation level for sub-frame sf-4.

n_overlap_lsf – One bit indicating whether LSF coefficients overlap from the previous definition in channel ch.

enable_ps_header – One bit indicating whether PS header information is present. If set to %1, the PS header data configuring the PS decoder is transmitted. Otherwise, the latest configuration persists.

enable_iid – One bit denoting the presence of IID parameters. If enable_iid is set to %1, Inter-channel Intensity Difference (IID) parameters will be sent from this point on in the bit stream. If enable_iid==%0, no IID parameters will be sent from this point on in the bit stream.

iid_mode - The configuration of IID parameters (number of bands and quantisation grid, iid_quant) is determined by iid_mode. Eight different configurations for IID parameters are supported (see Table 8.24).

Table 8.24 — IID mode configurations

iid_mode	nr_iid_par_tab	nr_ipdopd_par_tab	iid_quant	Index range
0 (000)	10	5	0	-7…7
1 (001)	20	11		-7…7
2 (010)	34	17		-7…7
3 (011)	10	5	1	-15…15
4 (100)	20	11		-15…15
5 (101)	34	17		-15…15
6 (110)	reserved			
7 (111)	reserved			

If no IID data is sent in the bit-stream, all IID parameters are reset to 0 (i.e. index=0).

The default and the fine quantization grids for IID, iid_quant = %0 and iid_quant = %1 are as provided in Table 8.B.18 and Table 8.B.17 respectively.

Table 8.25 — Default quantization grid for IID.

Index	-7	-6	-5	-4	-3	-2	-1	0
IID [dB]	-25	-18	-14	-10	-7	-4	-2	0
Index	1	2	3	4	5	6	7	
IID [dB]	2	4	7	10	14	18	25	

Table 8.26 — Fine quantization grid for IID.

Index	-15	-14	-13	-12	-11	-10	-9	-8
IID [dB]	-50	-45	-40	-35	-30	-25	-22	-19
Index	-7	-6	-5	-4	-3	-2	-1	0
IID [dB]	-16	-13	-10	-8	-6	-4	-2	0
Index	1	2	3	4	5	6	7	8
IID [dB]	2	4	6	8	10	13	16	19
Index	9	10	11	12	13	14	15	
IID [dB]	22	25	30	35	40	45	50	

Note that the configuration of the Inter-channel Phase Difference (IPD) / Overall Phase Difference (OPD) parameters, is strictly coupled to the IID configuration. This is also illustrated in Table 8.24.

enable_icc – One bit denoting the presence of ICC parameters. If enable_icc is set to %1, Inter-channel Coherence (ICC) parameters will be sent from this point on in the bit stream. If enable_icc==%0, no ICC parameters will be sent from this point on in the bit stream.

icc_mode – The configuration of Inter-channel Coherence parameters (number of bands and quantisation grid) is determined by icc_mode. Eight different configurations are supported for IC parameters (see Table 8.27).

Table 8.27 — ICC mode configuration

icc_mode	nr_icc_par_tab	Index range	Mixing procedure
0 (000)	10	0…7	R_a
1 (001)	20	0…7	
2 (010)	34	0…7	
3 (011)	10	0…7	R_b
4 (100)	20	0…7	
5 (101)	34	0…7	
6 (110)	reserved		
7 (111)	reserved		

If no ICC data is sent in the bit-stream, all ICC parameters are reset to 1 (i.e. index=0). The default quantization grid for ICC is provided in Table 8.28.

Table 8.28 — Quantization grid for ICC.

index	0	1	2	3	4	5	6	7
ρ	1	0.937	0.84118	0.60092	0.36764	0	-0.589	-1

enable_ext – The PS extension layer is enabled using the enable_ext bit. If it is set to %1 the IPD and OPD parameters are sent. If it's disabled, i.e. %0, the extension layer is skipped.

frame_class – The frame_class bit determines whether the parameter positions of the current frame are uniformly spaced accross the frame (FIX_BORDERS: frame_class==%0) or they are defined using the positions described by border_position (VAR_BORDERS: frame_class==%1).

num_env_idx – The number of (sets of) parameters (envelopes) per frame is determined using num_env_idx. In case of fixed parameter spacing (frame_class==%0) and a variable parameter spacing (frame_class==%1) this relation is shown in Table 8.29.

num_env – Local variable denoting the number of stereo envelopes (sets of parameters). num_env==0 signals that no new stereo parameters are transmitted and that the last parameters in the previous ps_data() element shall be kept unchanged and applied to the current ps_data() element.

Table 8.29 — Number of parameter sets num_env as a function of num_env_idx in case of fixed and variable spacing.

	num_env_tab[frame_class][num_env_idx]	
num_env_idx	frame_class==0	frame_class==1
0	0	1
1	1	2
2	2	3
3	4	4

border_position[e] – In case of variable parameter spacing the parameter positions are determined by border_position[e]. It contains the QMF sample index n_e for parameter set e of the current ps_data() element.

iid_dt[e] – This flag describes for envelope index e, whether the IID parameters are coded differentially over time (iid_dt==%1) or over frequency (iid_dt==%0). In the case iid_mode is different from the previous envelope (e-1), iid_dt[e] shall have the value 0% forcing frequency differential coding.

iid_data() – syntactic element containing IID data.

icc_dt[e] – This flag describes for envelope index e, whether the ICC parameters are coded differentially over time (icc_dt==%1) or over frequency (icc_dt==%0). In the case icc_mode is different from the previous envelope (e-1), icc_dt[e] shall have the value 0% forcing frequency differential coding.

icc_data() – syntactic element containing ICC data.

cnt – Local variable denoting the number of bytes used for the ps_extension() element.

ps_extension_size – The length of the PS extension layer is ps_extension_size, measured in bytes. If the extension size makes use of the escape code (ps_extension_size == 15) the length of the extension layer is extended by an additional amount of bytes.

esc_count – In case the escape code (ps_extension_size == 15) is used, esc_count describes the additional length of the PS extension layer measured in bytes.

num_bits_left – Local variable describing the number of bits left for reading in the ps_extension() element.

ps_extension_id – The identification tag (version) of the PS extension layer is given by ps_extension_id. Currently only one version is supported (see Table 8.30).

Table 8.30 — Description of ps_extension_id

ps_extension_id	Version
00 (0)	v0
01 (1)	reserved
10 (2)	reserved
11 (3)	reserved

fill_bits – These fill_bits accomplish byte-alignment of the ps_extension() data.

enable_ipdopd – The application of IPD and OPD parameters in the bit-stream is denoted by enable_ipdopd. If set (enable_ipdopd==%1) IPD and OPD parameters are sent, if disabled (enable_ipdopd==%0) no IPD and OPD parameters are sent for the current frame in the bit-stream. The quantization grid for both IPD and OPD is provided in Table 8.31. If no IPD or OPD data is sent in the bit-stream, all IPD and OPD parameters are set to 0 (i.e. index=0).

Table 8.31 — Quantization grid for IPD/OPD.

Index	0	1	2	3	4	5	6	7
Representation level	0	$\frac{\pi}{4}$	$\frac{\pi}{2}$	$\frac{3\pi}{4}$	π	$\frac{5}{4}\pi$	$\frac{3}{2}\pi$	$\frac{7}{4}\pi$

ipd_dt[e] – This flag describes for envelope index e, whether the IPD parameters are coded differentially over time (ipd_dt==%1) or over frequency (ipd_dt==%0). In the case iid_mode is different from the previous envelope (e-1), ipd_dt[e] shall have the value 0% forcing frequency differential coding.

ipd_data() – syntactic element containing IPD data.

opd_dt[e] – This flag describes for envelope index e, whether the OPD parameters are coded differentially over time (opd_dt==%1) or over frequency (opd_dt==%0). In the case iid_mode is different from the previous envelope (e-1), opd_dt[e] shall have the value 0% forcing frequency differential coding.

opd_data() – syntactic element containing OPD data.

reserved_ps – This bit is reserved and has a value %0.

iid_par_dt[e][b] – In case of differential coding of IID parameters over time (iid_dt[e]==%1), iid_par_dt[e][b] describes the IID index difference with respect to the b^{th} parameter position for envelope e-1. If no previous parameter is available, iid_par_dt[e][b] represents the IID index difference with respect to the decoded value 0 (i.e. index=0). The IID index iid_par[e][b] is determined as:

$$\text{iid\_par}[e][b] = \text{iid\_par}[e-1][b] + \text{iid\_par\_dt}[e][b]$$

where iid_par[e-1][b] represents the IID index of the previous envelope, e-1. The IID value *iid*[*b*] is obtained by using iid_par[e][b] as an index to Table 8.25 or Table 8.26, depending on iid_mode.

iid_par_df[e][b] – In case of differential coding of IID parameters over frequency (iid_dt[e]==%0), iid_par_df[e][b] describes the IID difference with respect to the $(b-1)^{th}$ parameter in envelope e. If no previous parameter is available, iid_par_df[e][b] represents the IID difference with respect to the decoded value 0 (i.e. index=0). The IID index iid_par[e][b] is determined as:

$$\text{iid\_par}[e][0] = \text{iid\_par\_df}[e][0]$$

$$\text{iid\_par}[e][b] = \text{iid\_par}[e][b-1] + \text{iid\_par\_df}[e][b] \qquad \text{for} \quad b > 0$$

where iid_par[e][b-1] represents the IID index of the previous IID value for envelope e. The IID value *iid*[*b*] is obtained by using iid_par[e][b] as an index to Table 8.25 or Table 8.26, depending on iid_mode.

icc_par_dt[e][b] – In case of differential coding of ICC parameters over time (icc_dt[e]==%1), icc_par_dt[e][b] describes the difference with respect to the b^{th} parameter position for envelope e-1. If no previous parameter is available, icc_par_dt[e][b] represents the ICC difference with respect to the decoded value 1 (i.e. index=0). The ICC index icc_par[e][b] is determined as:

$$\text{icc\_par}[e][b] = \text{icc\_par}[e-1][b] + \text{icc\_par\_dt}[e][b]$$

where icc_par[e-1][b] represents the ICC index of the previous envelope, e-1. The ICC value $\rho[b]$ is obtained by using icc_par[e][b] as an index to Table 8.28.

icc_par_df[e][b] – In case of differential coding of ICC parameters over frequency (icc_dt[e]==%0), icc_par_df[e][b] describes the ICC difference with respect to the $(b-1)^{th}$ parameter for envelope e. If no previous parameter is available, icc_par_df[e][b] represents the ICC difference with respect to the decoded value 1 (i.e. index=0). The ICC index icc_par[e][b] is determined as:

$$\text{icc\_par}[e][0] = \text{icc\_par\_df}[e][0]$$
$$\text{icc\_par}[e][b] = \text{icc\_par}[e][b-1] + \text{icc\_par\_df}[e][b] \qquad \text{for} \quad b > 0$$

where icc_par[e][b-1] represents the ICC index of the previous ICC value for envelope e. The ICC value $\rho[b]$ is obtained by using icc_par[e][b] as an index to Table 8.28.

ipd_par_dt[e][b] – In case of differential coding of IPD parameters over time (ipd_dt[e]==%1), ipd_par_dt[e][b] describes the IPD difference with respect to the b^{th} parameter position for envelope e. If no previous parameter is available, ipd_par_dt[e][b] represents the IPD difference with respect to the decoded value 0 (i.e. index=0). Note: for IPD parameters modulo-8 differential coding is applied. The IPD index ipd_par[e][b] is determined as:

$$\text{ipd\_par}[e][b] = \text{mod}(\text{ipd\_par}[e-1][b] + \text{ipd\_par\_dt}[e][b], 8)$$

where ipd_par[e-1][b] represents the IPD index of the previous envelope, e-1. The IPD value $ipd[b]$ is obtained by using ipd_par[e][b] as an index to Table 8.31.

ipd_par_df[e][b] – In case of differential coding of IPD parameters over frequency (ipd_dt[e]==%0), ipd_par_df[e][b] describes the IPD difference with respect to the $(b-1)^{th}$ parameter for envelope e. If no previous parameter is available, ipd_par_df[e][b] represents the IPD difference with respect to the decoded value 0 (i.e. index=0). Note: for IPD parameters modulo-8 differential coding is applied. The IPD index ipd_par[e][b] is determined as:

$$\text{ipd\_par}[e][0] = \text{ipd\_par\_df}[e][0]$$
$$\text{ipd\_par}[e][b] = \text{mod}\left(\text{ipd\_par}[e][b-1] + \text{ipd\_par\_df}[e][b], 8\right) \qquad \text{for} \quad b > 0$$

where ipd_par[e][b-1] represents the IPD index of the previous IPD value for envelope e. The IPD value $ipd[b]$ is obtained by using ipd_par[e][b] as an index to Table 8.28.

opd_par_dt[e][b] – In case of differential coding of OPD parameters over time (opd_dt[e]==%1), opd_par_dt[e][b] describes the OPD difference with respect to the b^{th} parameter position for envelope (e-1). If no previous parameter is available, opd_par_dt[e][b] represents the OPD difference with respect to the decoded value 0 (i.e. index=0). Note: for OPD parameters modulo-8 differential coding is applied. The OPD index opd_par[e][b] is determined as:

$$\text{opd\_par}[e][b] = \text{mod}(\text{opd\_par}[e-1][b] + \text{opd\_par\_dt}[e][b], 8)$$

where opd_par[e-1][b] represents the OPD index of the previous envelope, e-1. The OPD value $opd[b]$ is obtained by using opd_par[e][b] as an index to Table 8.31.

opd_par_df[e][b] – In case of differential coding of OPD parameters over time (opd_dt[e]==%0), opd_par_dt[e][b] describes the OPD difference with respect to the $(b-1)^{th}$ parameter position for envelope e. If

no previous parameter is available, opd_par_df[e][b] represents the OPD difference with respect to the decoded value 0 (i.e. index=0). Note: for OPD parameters modulo-8 differential coding is applied. The OPD index opd_par[e][b] is determined as:

$$\text{opd\_par}[e][0] = \text{opd\_par\_df}[e][0]$$
$$\text{opd\_par}[e][b] = \text{mod}\left(\text{opd\_par}[e][b-1] + \text{opd\_par\_df}[e][b], 8\right) \qquad \text{for} \quad b > 0$$

where opd_par[e][b-1] represents the OPD index of the previous OPD value for envelope e. The OPD value *opd*[*b*] is obtained by using opd_par[e][b] as an index to Table 8.28.

8.5.3 Indexing of sub-frames

In the case differential coding is applied from one sub-frame to the next, a negative sub-frame index sf of frame *k* might be indexed. In case this occurs, the negative sub-frame shall be corrected according to

sf = sf + nrof_subframes.

Note that the sub-frame index thus obtained resides in frame *k-1*. Similarly, in the case that sf is larger than nrof_subframes, the sub-frame shall be corrected according to

sf = sf – nrof_subframes.

The sub-frame index thus obtained resides in frame *k+1*.

8.6 Decoding processes

The parametric stereo decoder is illustrated in Figure 8.1. After de-formatting the bit-stream, the monaural signal M is reconstructed as the combination of transients, sinusoids and noise. Subsequently the stereo parameters are used to reconstruct the left and right signal from the monaural encoded signal. For dual-mono and mono, the parametric stereo decoder is not used.

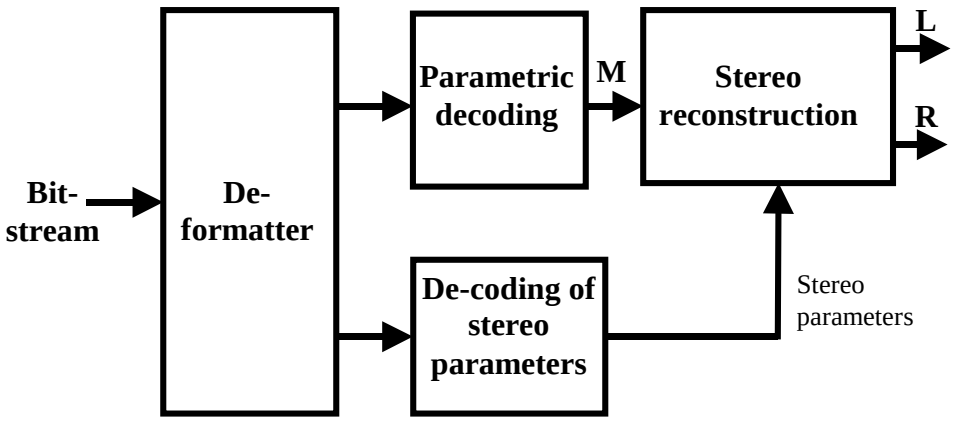


Figure 8.1 — Illustration of the integration of the stereo coding tools into the parametric decoder

The parametric decoder consists of three decoders: the transient decoder (8.6.1), the sinusoidal decoder (8.6.2) and the noise decoder (8.6.3). The decoded signal is obtained by summing the output of the three decoders per frame per channel. In the description of the decoders in the parametric decoder, the indexing of the sub-frame sf and channel ch is occasionally omitted for clarity.

8.6.1 Transients

Two types of transients are defined, a step transient and a transient of the Meixner type. The decoding of the step transient only involves the interpretation of the position. For the Meixner type, a parameterized envelope *g*[*n*] and a number of sinusoids need to be decoded.

8.6.1.1 Step transient

A step transient does not generate a signal of its own (as the Meixner transient does), but it is used to modify the window shape for synthesizing sinusoidal and noise components (see sections 8.6.2.4 and 8.6.3).

8.6.1.2 Meixner transient

For the decoding of the Meixner transient, first the envelope needs to be generated. For the envelope the following parameters are required: the start position t_{loc}, the onset slope indicated by t_b_par (*b* parameter) and the decaying slope represented by t_chi_par (ξ parameter).

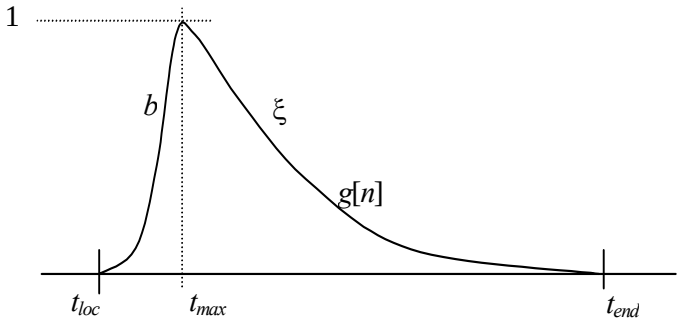


Figure 8.2 — The Meixner envelope is described by the function *g*[*n*]. The attack slope is controlled by the parameter *b*. The decay is controlled by the parameter ξ

The start time of the envelope, t_{loc}, is referred to as $n = 0$ for ease of explanation. This envelope *g*[*n*] is generated according to

$$g[0] = \frac{\left(1-\xi^2\right)^{\frac{b}{2}}}{a_{\max}},$$

$$g[n] = g[n-1]\cdot\xi\cdot\sqrt{\frac{b+n-1}{n}},$$

for n=1 up to and including t_{end}. The end position of the transient window t_{end} is defined below. The maximum a_{max} at position t_{max} is given by the approximations

$$t_{\max} = \frac{b-1}{-2\cdot\log_e(\xi)},$$

$$a_{\max} = \sqrt{\frac{-2\cdot\log_e(\xi)}{\Gamma(b)}}\cdot\left(\frac{b-1}{e}\right)^{\frac{b-1}{2}}.$$

These complex expressions, especially that for a_{max}, have been evaluated for the allowed values of t_b_par and t_chi_par, and are tabulated below.

Table 8.32 — t_{max} for all possible values of t_b_par and t_chi_par.

t_{max}		t_b_par			
		0	1	2	3
t_chi_par	0	15	30	45	59
	1	20	39	57	75
	2	30	56	79	100
	3	57	96	126	150

Table 8.33 — a_{max} for all possible values of t_b_par and t_chi_par.

a_{max}		t_b_par			
		0	1	2	3
t_chi_par	0	0.152713500109658	0.131630525645664	0.120142673294398	0.112550174511598
	1	0.132843681407528	0.115639700421076	0.106510539071702	0.100663024431527
	2	0.109279971016712	0.0971964875412947	0.0909719057150294	0.0872632874594248
	3	0.0797175749717262	0.0744985442180281	0.0723059623257423	0.0715041477354716

The transient needs to be synthesized until t_{end}. This position t_{end} is defined as the end of the second complete subsequent sub-frame after the transient position t_{loc}. This is illustrated in Figure 8.3.

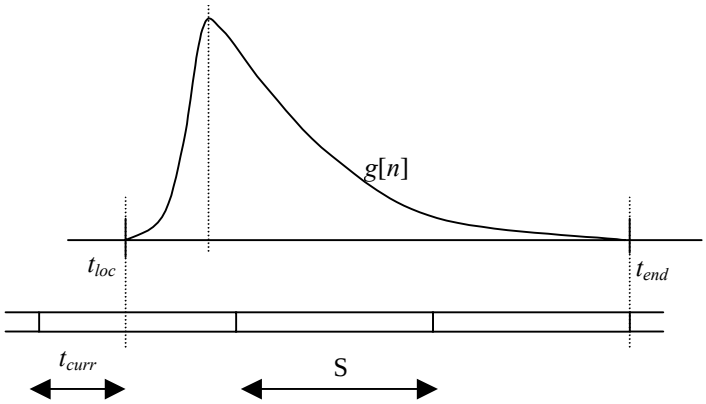


Figure 8.3 — The end position t_end is located at the end of the second complete sub-frame after the transient position t_loc

$$t_{end} = 3 \cdot \mathrm{S} - t_{loc} - 1$$

Note that the counting of samples in a sub-frame starts at 0. If the transient starts exactly at the beginning of a sub-frame (t_{loc} = 0), then t_{end} = 3*S-1.

8.6.1.3 Sinusoids under envelope

The resulting representation of the transient is obtained by combining the envelope and the sinusoids according to

$$t[n] = g[n] \cdot \sum_{i=1}^{t\_nrof\_\sin} ta_q[i] \cdot \cos(tf_q[i] \cdot n + tp_q[i]),$$

for n=0 up to and including t_{end}.

8.6.2 Sinusoids

8.6.2.1 Linking

A refresh frame, indicated by refresh_sinusoids == %1, is used to indicate whether the frame is starting with absolute values for all continuations or whether the frame starts with differentially coded continuations. For the birth of each sinusoidal track in a frame, s_cont is supplied in the bit-stream to signal the number of sub-frames the track continues after the current sub-frame in this frame. In case the track continues in the first sub-frame of the next frame, 1 is added to that number. If the track continues beyond the first sub-frame of the next frame, 2 is added to that number. Based on this information, the decoder is implicitly able to link the parameters that belong to a track.

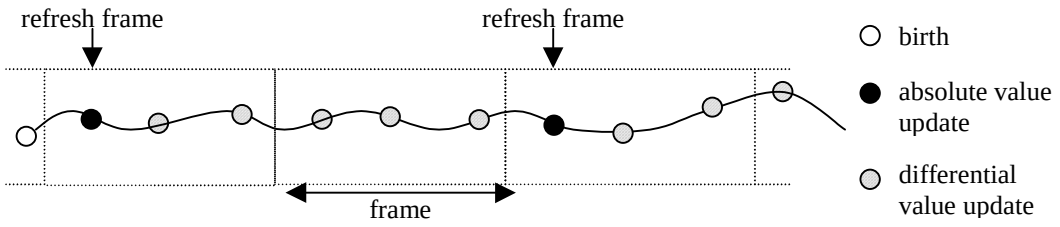


Figure 8.4 — Example for updating of a track. For every other frame, the track continuations are updated by their absolute values. In the case decoding starts at refresh_sinusoids==%0, the differential updates will be ignored until a refresh_sinusoids==%1 or a birth is received

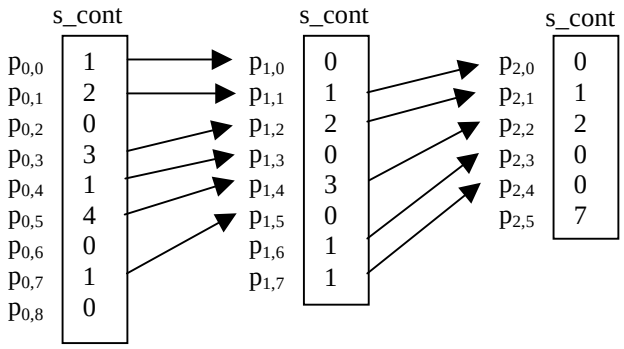


Figure 8.5 — Illustration of how the signaling of continuations operates. Parameter $p_{i,j}$ indicates the parameters (f, a and φ) of the j^{th} sinusoidal component in the i^{th} sub-frame. The information s_ cont that is sent along with these parameters provides the information on the evolution of the corresponding component in the next (sub)frame. In the above example, in sub-frame 1, components $p_{1,0}$, $p_{1,1}$, $p_{1,2}$, $p_{1,3}$, $p_{1,4}$ and $p_{1,5}$ are continued from sub-frame 0. Components $p_{1,0}$, $p_{1,3}$ and $p_{1,5}$ will die in sub-frame 1 and components $p_{1,6}$ and $p_{1,7}$ are new births

s_ cont is filled in the following order:

1) Continuations

2) Births (sorted on frequency in ascending order)

When going from one sub-frame to the next, the decoder keeps track of the number of continuations, s_nrof_continuations[sf]. The number of continuations present in sub-frame sf+1 can be directly derived from the number of entries in s_cont[sf] unequal to zero. For the first sub-frame of a frame, s_nrof_continuations is read from the bit-stream, to enable random access.

The total amount of sinusoidal components in sub-frame sf, s_nrof_sin[sf], is calculated as

sf == 0:

$$s\_nrof\_sin[0] = s\_nrof\_continuations[0][ch] + s\_nrof\_births[0][ch],$$

sf > 0:

$$s\_nrof\_sin[sf] = \sum_{i=0}^{max\_nrof\_sinusoids-1} (s\_cont[sf-1][ch][i] > 0) + s\_nrof\_births[sf][ch].$$

8.6.2.2 Decoding of sinusoidal parameters

In the description below, we assume to have a sinusoidal track of length κ, in sub-frames *sf* = [*K, K*+κ-*1*]. For births of the track (*sf* = *K*), the frequency and phase for sinusoid index *n* are represented by f_q[*K*][*ch*][*n*] and sp_q[*K*][*ch*][*n*] respectively. For continuations, in order to derive the frequency and phase information for a sub-frame, the representation levels together with tracking information is required. Figure 8.6 shows the decoder structure for continuations.

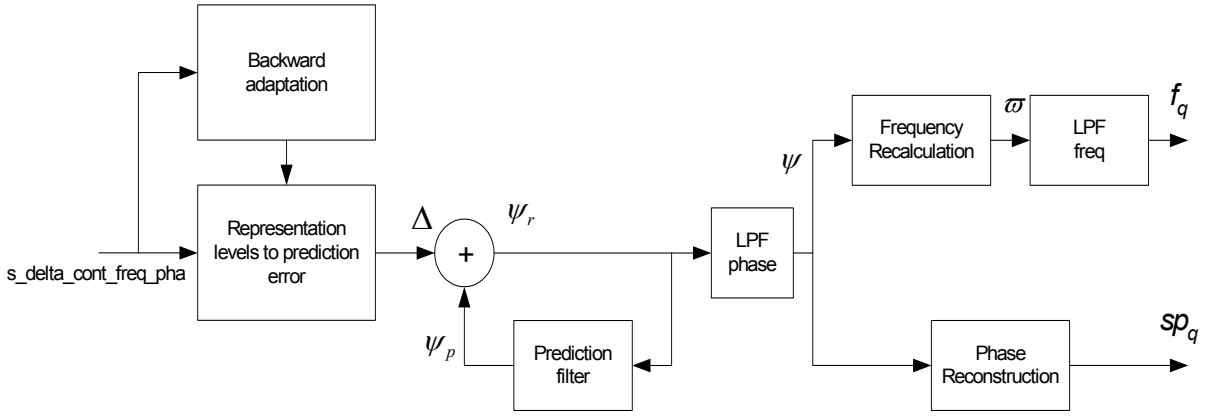


Figure 8.6 — The phase/frequency decoding system for continuations

For a continuation (*sf* = [*K*+*1, K*+κ-*1*]), the representation levels s_delta_cont_freq_pha[sf][ch][n] are converted to a quantized prediction error Δ[*sf*][*ch*[*n*] using Table 8.35 with index == 2. The quantized prediction error Δ[*sf*][*ch*][*n*] is subsequently multiplied by scale factor *c*[*sf*][*ch*][*n*]. For the first continuation, *c*[*K+1*][*ch*][*p*], where *p* represents the sinusoid index in sub-frame *K*+1, depends on the frequency of the birth, i.e. f_q[*K*]. Table 8.34 shows the value of the scale factor *c* for the possible frequency ranges of f_q[*K*].

Table 8.34 — Scale factor table

Freq. range (in Hertz)	Scale factor *c*[*K*+1]
[0 – 500] Hz	1/8
<500-1000] Hz	1/4
<1000-4000] Hz	1/2
<4000-22050] Hz	1

For other continuations, *sf* = [*K+2, K*+κ-*1*], *c* is modified according to the received representation levels along the track by means of the block "Backward Adaptation". If s_delta_cont_freq_pha[sf][ch][n] is either 1 or 2 (inner level) for sub-frame *sf*, then *c* for sub-frame *sf+1* is set to

$$c[sf+1][ch][p] = c[sf][ch][n] \cdot 2^{-1/4}.$$

If s_delta_cont_freq_pha[sf][ch][n] equals 0 or 3 (outer level), then *c* for sub-frame *sf+1* is set to

$$c[sf+1][ch][p] = c[sf][ch][n] \cdot 2^{1/2}.$$

In order to avoid very small or very large entries in prediction error, the adaptation is only done if the absolute value of the inner level, 0.75*c*[*sf+1*][*ch*][*p*], is between $\pi/128$ and $3\pi/8$. In the case were the inner level is less than or equal to $\pi/128$ or greater than or equal to $3\pi/8$, the scale factor *c*[*sf+1*][*ch*][*p*] is set to *c*[*sf*][*ch*][*n*]. This is illustrated in Table 8.35.

After having obtained the quantised prediction error Δ, the output of the prediction filter is added to it, resulting in the unwrapped phase ψ_r.

$$\psi_r[sf][ch][n] = \psi_p[sf][ch][n] + \Delta[sf][ch][n] \cdot c[sf][ch][n].$$

A second-order predictor is used. The input-output behaviour of the filter is

$$\psi_p[sf+1][ch][p] = 2 \cdot \psi_r[sf][ch][n] - \psi_r[sf-1][ch][q],$$

where *q* is the sinusoid index in sub-frame *sf*-1, ψ_r is the input and ψ_p is the output of the prediction filter.

To initialize the prediction filter we need one value of history for the input ψ_r[*K-1*], where *K* is the sub-frame index of the birth of the track. Since this value is not available, we assume that the frequency is constant at sub frame *(K-1)*. For birth of a track we have the frequency and phase information available, so we can calculate the input at *(K-1) and K* according to,

$$\psi_r[K-1][ch][n] = sp_q[K][ch][n] - f_q[K][ch][n] \cdot S,$$
$$\psi_r[K][ch][n] = sp_q[K][ch][n].$$

where *S* represents the update interval. The unwrapped phases are low-pass filtered by block LPF-phase. This is done as follows:

$$\psi[sf][ch][n] = 0.25 \cdot \psi_r[sf+1][ch][p] + 0.5 \cdot \psi_r[sf][ch][n] + 0.25 \cdot \psi_r[sf-1][ch][q],$$

where *sf* is the sub-frame index along a track. At the end of a track (*sf* = *K*+κ-*1*), the following rule is applied:

$$\psi[sf][ch][n] = \psi_r[sf][ch][n].$$

The reconstructed phases are derived from the smoothed unwrapped phases as follows:

$$sp_q[sf][ch][n] = \mathrm{mod}(\psi[sf][ch][n] + \pi, 2\pi) - \pi.$$

To obtain the frequency, the unwrapped phases have to be differentiated along the track. The differentiation is implemented by an approximation. The frequency is obtained by:

$$\varpi[sf][ch][n] = \left(\psi[sf][ch][n] - \psi[sf-1][ch][q]\right)\frac{2}{S} - \varpi[sf-1][ch][q],$$

where S is the update interval and $\varpi[K][ch][n] = f_q[K][ch][n]$. As the birth phase and birth frequency are known in the decoder, the frequencies ϖ of the subsequent frames are calculated. In order to attenuate the noisy signal that is introduced by this differentiation, a low-pass filter is applied on the frequencies (LPF-freq):

$$f_q[sf][ch][n] = 0.25 \cdot \varpi[sf+1][ch][p] + 0.5 \cdot \varpi[sf][ch][n] + 0.25 \cdot \varpi[sf-1][ch][q].$$

For the first continuation of a track (*sf=K+1*), the definition is changed to:

$$f_q[K+1][ch][p] = 0.5 \cdot \varpi[K+1][ch][p] + 0.5 \cdot \varpi[K][ch][n].$$

Also the last frequency in the track (*sf* = *K*+κ-*1*) is obtained in a different manner:

$$f_q[sf][ch][n] = 0.5 \cdot \varpi[sf][ch][n] + 0.5 \cdot \varpi[sf-1][ch][q].$$

For tracks of length κ=2 the continuation is calculated according to

$$f_q[sf][ch][n] = 0.5 \cdot \varpi[sf-1][ch][q] + 0.5 \cdot \varpi[sf][ch][n].$$

In this way, the phases and frequencies are obtained from the representation levels s_delta_cont_freq_pha.

In refreshes frames, the following procedure applies. If *sf*=[*K, …, K+R, …, K+κ-1*]. The sub-frame, *K+R* is the first sub-frame of a frame with refresh_sinusoids == %1. Sub-frame (*K+R*-1) is the last sub-frame of a frame with refresh_sinusoids_next_frame == %1. The phase and frequency values for sub-frame *K* up to and including sub-frame *K+R*-1 are obtained as described above, as if the track ends at sub-frame (*K+R*-1). The phase and frequency values for sub-frame *K+R* up to and including sub-frame (*K*+κ-1) are obtained as described above, as if sub-frame *K+R* is a birth. For the initialization of the quantized prediction error Δ, s_adpcm_grid is used as index to Table 8.35 and *c*[*K+R*][*ch*][*n*] = 1.

Table 8.35 — Quantized prediction error Δ

Δ	s_delta_cont_freq_pha			
index	0	1	2	3
0	-4.2426	-1.0607	1.0607	4.2426
1	-3.5676	-0.8919	0.8919	3.5676
2	-3.0000	-0.7500	0.7500	3.0000
3	-2.5227	-0.6307	0.6307	2.5227
4	-2.1213	-0.5303	0.5303	2.1213
5	-1.7838	-0.4460	0.4460	1.7838
6	-1.5000	-0.3750	0.3750	1.5000
7	-1.2613	-0.3153	0.3153	1.2613
8	-1.0607	-0.2652	0.2652	1.0607
9	-0.8919	-0.2230	0.2230	0.8919
10	-0.7500	-0.1875	0.1875	0.7500
11	-0.6307	-0.1577	0.1577	0.6307
12	-0.5303	-0.1326	0.1326	0.5303
13	-0.4460	-0.1115	0.1115	0.4460
14	-0.3750	-0.0938	0.0938	0.3750
15	-0.3153	-0.0788	0.0788	0.3153
16	-0.2652	-0.0663	0.0663	0.2652
17	-0.2230	-0.0557	0.0557	0.2230
18	-0.1875	-0.0469	0.0469	0.1875
19	-0.1577	-0.0394	0.0394	0.1577
20	-0.1326	-0.0331	0.0331	0.1326
21	-0.1115	-0.0279	0.0279	0.1115

Note that in the case phase_jitter_present is set to %1 and phase and frequency are obtained through ADPCM decoding, no phase jitter is applied.

8.6.2.3 Synthesis of sinusoids for segments without a transient

The sinusoidal parameters are used to synthesize the sinusoidal components. This is done on a segment basis, a segment consisting of L samples.

Synthesis uses a 50% overlap and add strategy. In order to synthesize a sub-frame, the parameters of the previous sub-frame need to be available at the start of a new frame. This means that the parameters of the last sub-frame in the previous frame need to be available.

The actual synthesis of a sinusoid is according to

$$s_i[n] = sa_q[i] \cdot \cos\left(f_q[i] \cdot \left(n - \frac{L-1}{2} \right) + sp_q[i] \right), \quad n = [0, L-1]$$

Note that the phase information sp_q is defined for the middle of the segment (=(L-1)/2). For original phase, sp_q is calculated from s_phi (see section 8.5.2). For continuations, the phase is calculated as described in section 8.6.2.2.

In overlap and add the following, amplitude complementary symmetric window is chosen

$$w_s[n] = 1/2 - 1/2 \cos\left(\frac{\pi(2n+1)}{L} \right), \quad n = [0, L-1].$$

A segment of length L is obtained by

$$\tilde{s}_{sf}[n] = w_s[n] \sum_{i=0}^{s\_nrof\_\sin[sf]} s_i[n], \quad n = [0, L-1]$$

The sinusoidal contribution for sub-frame sf is then computed using the contribution from the previous sub-frame according to

$$s_{sf}[n] = \tilde{s}_{sf-1}[\mathrm{S}+n] + \tilde{s}_{sf}[n], \quad n = [0, \mathrm{S}-1]$$

8.6.2.4 Synthesis of sinusoids for segments without a transient

For sinusoids with a frequency lower than 400Hz only the window $w_s[n]$ defined in the previous section should be used, even at a segment containing a transient.

For other sinusoids, different window shapes apply for synthesis of a segment, depending on the course of a component (continuation, birth, or death). As a result of transient positions, some of these windows are modified. All the possibilities are illustrated in Figure 8.7.

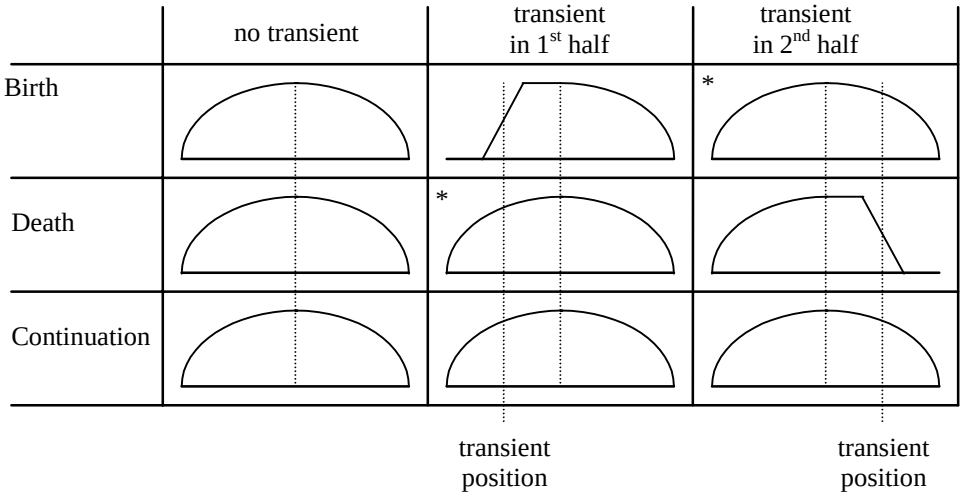


Figure 8.7 — Possible synthesis windows $w_s[n]$ for births, deaths and continuations, dependent on transient occurrence and position. For the situations denoted by * the transient and sinusoidal component have no relation.

This figure shows that there are two situations where the default window shape may not be used:

- a birth in combination with a transient located in the first half of the segment
- a death in combination with a transient in the second half of the segment.

8.6.2.4.1 Birth and transient in first half of segment

For births, a fade in window will be used to prevent substantial energy leaking into the interval prior to the transient position. In Figure 8.7 only a coarse shape of the window is shown. Figure 8.8 shows a more detailed view of the window.

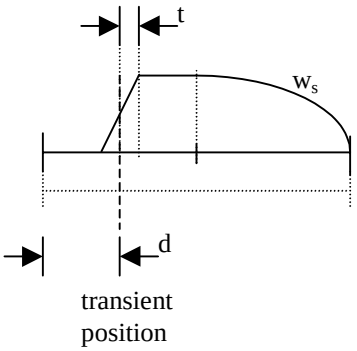


Figure 8.8 — Synthesis window shape for a birth where a transient occurs in the first half of the segment

The following expression exactly describes the window $w_s[n]$

$$w_s[n] = \begin{cases} 0 & ,0 \le n < d-10 \\ \dfrac{n-d+11}{22} & ,d-10 \le n \le d+10 \\ 1 & ,d+10 < n < S \\ \frac{1}{2} - \frac{1}{2}\cos\left(\pi \cdot \dfrac{2n+1}{L}\right) & ,S \le n < L \end{cases}$$

In the special case that the transient position is located within 10 samples from the “edge”, the slope is moved (see Figure 8.9).

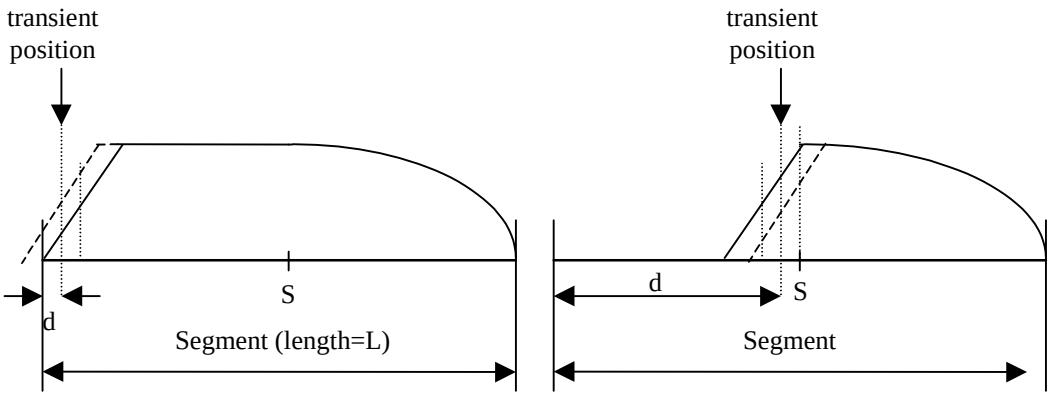


Figure 8.9 — Illustration of the fade-in window in the cases where the transient is located less than 10 samples from the “edge”; the slope is shifted to start at the “edge” (left picture) or end at the “edge” (right picture)

For the left picture in Figure 8.9 this means that if $0 \le d < 10$, then use $d = 10$ for determining the window function $w_s[n]$. For the right picture in Figure 8.9 this means that if $S\text{-}10 \le d < S$, then use $d = S–11$ for determining the window function $w_s[n]$.

8.6.2.4.2 Death and transient in second half of segment

In Figure 8.7 only a coarse shape of the window is shown. Figure 8.10 shows a more detailed view of the window.

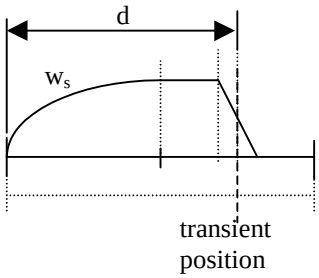


Figure 8.10 — Synthesis window shape for a death where a transient occurs in the second half of the segment

The following expression describes the window $w_s[n]$

$$w_s[n] = \begin{cases} \frac{1}{2} - \frac{1}{2}\cos\left(\pi \cdot \frac{2n+1}{L}\right) & ,0 \le n < S \\ 1 & ,S \le n < d-10 \\ \frac{d-n+11}{22} & ,d-10 \le n \le d+10 \\ 0 & ,d+10 < n < L \end{cases}.$$

In the special case that the transient position is located within 10 samples from the “edge”, the slope is moved (see Figure 8.11).

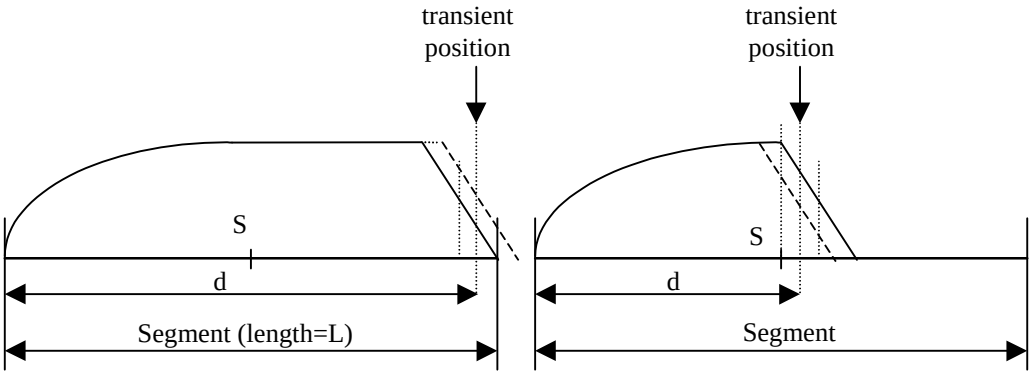


Figure 8.11 — Illustration of the fade-out window in the cases where the transient is located less than 10 samples from the “edge”; the slope is shifted to end at the “edge” (left picture) or start at the “edge” (right picture)

For the left picture in Figure 8.11 this means that if L-10 ≤ d < L , then use d = L-11 for determining the window function $w_s[n]$. For the right picture in Figure 8.11 this means that if S ≤ d < S + 10, then use d = S+10 for determining the window function $w_s[n]$.

8.6.3 Noise

The noise is synthesized in intervals of 4 sub-frames or 2L samples. The model for noise synthesis consists of a pseudo random number generator, a temporal envelope adjuster, a window mechanism for overlap and add and an IIR filter.

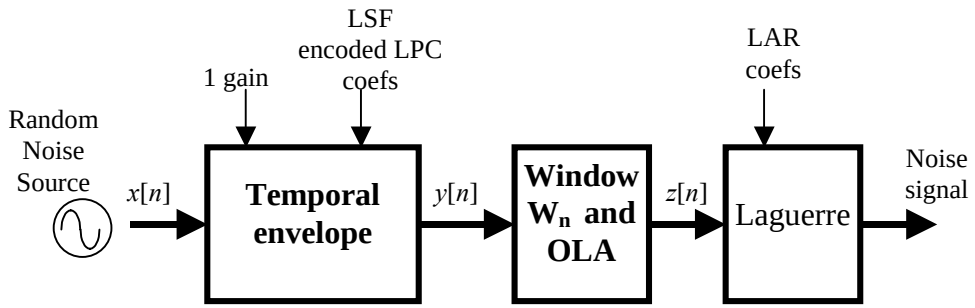


Figure 8.12 — Noise decoder. Time intervals of random noise excitations are adjusted by temporal envelopes which are windowed, overlap/added and subsequently filtered by the Laguerre filter, resulting in the noise signal. Per iteration one interval of noise is created for n = [0, 2L-1]

The temporal envelope *H*[*n*] is represented by using a single gain and a number of line spectral frequencies (LSFs) representing the LPC coefficients. Both the gain and the LSFs are updated once every 4 sub-frames (2L samples). The Laguerre coefficients are represented by LAR parameters. These are updated once per 2 sub-frames (L samples), which is twice as low update-rate as used for the temporal envelope, see Figure 8.13.

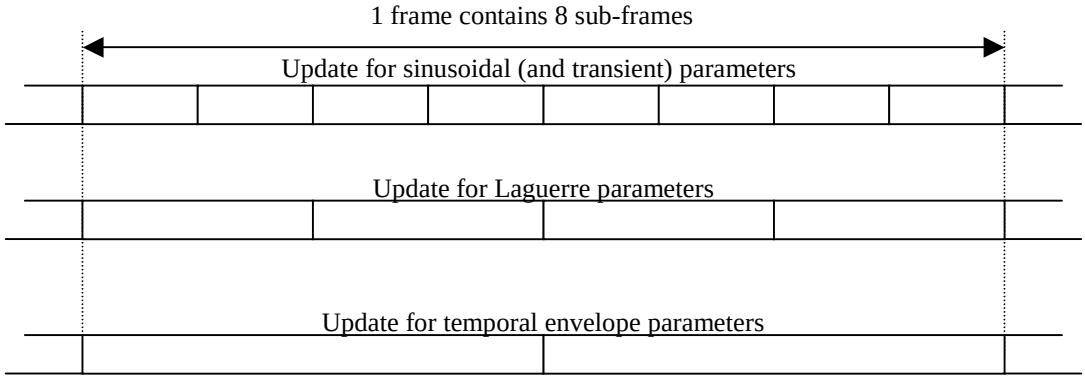


Figure 8.13 — The update rate for the Laguerre parameters is for every two sub-frames (2S = L). The temporal parameters are updated every four sub-frames (4S = 2L)

In order to prevent discontinuities, intervals that are modified by a temporal envelope have an overlap of 25%. In the overlap region a Hanning window is employed, see Figure 8.14. Note that the first 4 generated sub-frames of 2L samples start with a fade-in using the Hanning window.

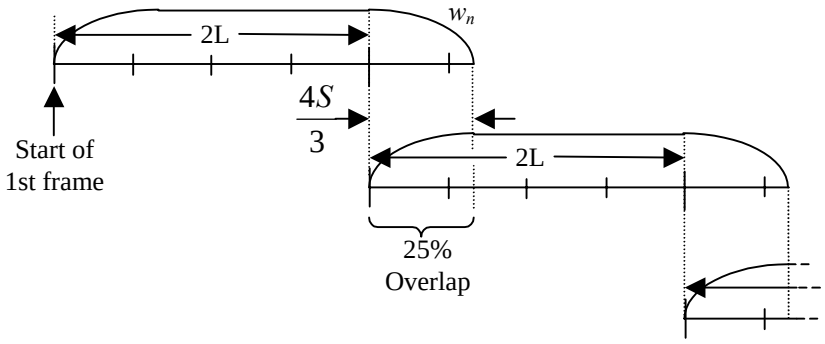


Figure 8.14 — Overlap and add. In the overlap a half Hanning window h is used. In the intermediate part the window is set to 1. The first 4 sub-frames are starting with a fade-in realised by the Hanning window

The window w_n is defined as given below.

$$w_n[n] = \begin{cases} \frac{1}{2} - \frac{1}{2}\cos\left(\frac{3\pi(2n+1)}{8S}\right) & , \quad n = [0, \frac{4S}{3} - 1], \\ 1 & , \quad n = [\frac{4S}{3}, 4S - 1], \\ w_n[\frac{16S}{3} - 1 - n] & , \quad n = [4S, \frac{16S}{3} - 1]. \end{cases}$$

The operations performed by each of the individual processes is discussed in more detail in the following sections.

8.6.3.1 Noise generation

The noise is generated by applying a pseudo random number generator, defined by a linear congruential sequence U

$$U[n+1] = \text{mod}\ (a*U[n]+c,\ m),$$

where $U[0]$ is the starting value, a the multiplier, c the increment and m the modulus (with $m=2^{32}$). At the start of decoding the starting value is set to channel_number (0 = left, 1 = right), resulting in independent noise sources for each channel. For the generation of each following interval of noise, the starting value is set to the end value of the previous interval. The algorithm is listed below.

```
#define RAND_SCALE      (1/4294967296.0)
#define RAND_FACTOR     1664525L
#define RAND_OFFSET     1013904223L
double  noiseUDN(unsigned long *lp_seed)
{
    *lp_seed = (*lp_seed * RAND_FACTOR+RAND_OFFSET) & 0xFFFFFFFF;
    return *lp_seed * RAND_SCALE;
}
```

This algorithm returns a value $U = [0,1\rangle$. A normal distribution X is obtained by addition of 12 consecutive samples of the distribution (U-0.5). For the next sample of X, 12 new consecutive samples are used. Using this normal distribution X, by means of noise filtering, the spectral noise is generated.

In order to avoid discontinuities in the noise generation, overlapping time intervals are taken from the free running noise generator. This is realized by copying the seed values from the previous interval to the current interval during overlap. This is illustrated in Figure 8.15.

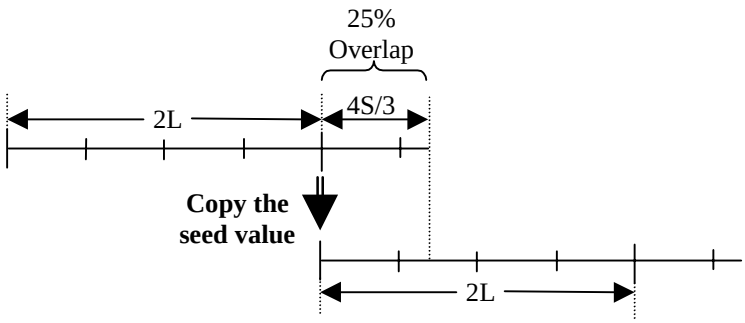


Figure 8.15 — The start-seed value used in the next interval is copied from the previous interval at the corresponding time location

8.6.3.2 Temporal envelope

The temporal envelope is applied to an interval of 2L+4S/3 samples, which are generated by the random noise generator. The temporal envelope shape is represented by the time domain equivalent of Line Spectral Frequencies, which are again a representation of LPC coefficients. In total n_nrof_lsf LSFs are encoded. An additional gain parameter is used to scale the entire envelope. Since the LSF intervals have an overlap of 25% there is a potential redundancy in LSF parameters in this overlap region. In the case this redundancy is present, only one set of LSFs, valid for two envelopes in the overlap region, is encoded. This situation is signalled by the parameter n_overlap_lsf. In the case of refresh_noise == %1, the first LSF and gain for that particular interval are encoded in absolute values by means of the parameters n_lsf and n_gain. Subsequent LSFs are encoded differentially w.r.t. each other, see Figure 8.16.

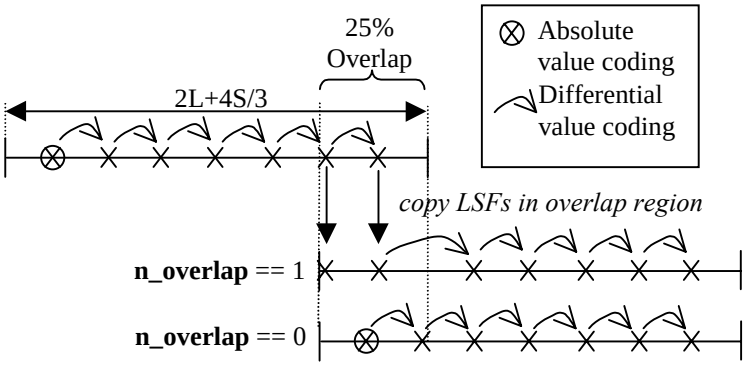


Figure 8.16 — Example for 7 LSF parameters. For the first interval, the first LSF parameter is encoded in its absolute value n_lsf. Subsequent LSF values in that frame are differentially encoded w.r.t. each other by means of n_delta_lsf. In the case n_overlap equals 1, all LSFs in the overlap region are copied to the next interval. Subsequent LSFs are again differentially encoded w.r.t. each other. In the case n_overlap equals 0, the LSFs are obtained similar to the first interval

In the case refresh_noise == %0, the gain parameter for that interval is encoded differentially w.r.t. the gain of the previous frame by means of the parameter n_delta_gain. The encoding of the LSF parameters in that situation depends on the setting of n_overlap_lsf. In the case refresh_noise == %0 and n_overlap_lsf == %0, the LSF are encoded as in the situation where refresh_noise == %1. In the case refresh_noise == %0 and n_overlap_lsf == %1, the number of LSF coefficients that overlap, n_nrof_overlap, is computed from the previous definition in channel ch according to

```
n_nrof_overlap = 0;
   for ( i = 0; i < n_nrof_lsf; i++) {
   if ( n_lsf[sf-4][ch][i] >= 192 ) n_nrof_overlap++
}
```

The LSF coefficients that overlap are copied from the previous definition according to

```
for ( i = 0, j = n_nrof_lsf – n_nrof_overlap; i < n_nrof_overlap; i++, j++)
{
   n_lsf[sf][ch][i] = n_lsf[sf-4][ch][j] - 192
}
```

8.6.3.2.1 Decoding the Gain and LSF parameters

The gain scales the entire temporal envelope. Since there are two envelopes per frame, the gain for the first and second temporal envelope are encoded in sf=0 and sf=4 respectively. The gain factor *G* which is actually applied to a temporal envelope is calculated as

$$G = \begin{cases} 0 & ngain_{rl} == 0 \\ 10^{\frac{ngain_{rl}-21}{20}} & \text{otherwise} \end{cases}.$$

The decoded LSFs, $nlsf_q$ are transformed to a-parameters using the following equations. First, all LSFs are transformed to positions on the unit circle

$$z[i] = e^{j \cdot nlsf_q[i]} \qquad i = [1, n\_nr\_of\_lsf].$$

These positions are then split into two polynomials:

$$z_P[i] = z[2i] \qquad i = \left[1, \left\lfloor \frac{n\_nr\_of\_lsf}{2} \right\rfloor\right],$$

$$z_Q[i] = z[2i-1] \qquad i = \left[1, \left\lceil \frac{n\_nr\_of\_lsf}{2} \right\rceil\right].$$

For both polynomials the complex conjugates are concatenated

$$z_P[i] = z_P^*\left[i - \left\lfloor \frac{n\_nr\_of\_lsf}{2} \right\rfloor\right] \qquad i = \left[1 + \left\lfloor \frac{n\_nr\_of\_lsf}{2} \right\rfloor, 2\left\lfloor \frac{n\_nr\_of\_lsf}{2} \right\rfloor\right],$$

$$z_Q[i] = z_Q^*\left[i - \left\lceil \frac{n\_nr\_of\_lsf}{2} \right\rceil\right] \qquad i = \left[1 + \left\lceil \frac{n\_nr\_of\_lsf}{2} \right\rceil, 2\left\lceil \frac{n\_nr\_of\_lsf}{2} \right\rceil\right].$$

The polynomials are calculated as following

$$p_P = \prod_{i=1}^{2\left\lfloor \frac{n\_nr\_of\_lsf}{2} \right\rfloor} \left(z - z_P[i]\right),$$

$$p_Q = \prod_{i=1}^{2\left\lceil \frac{n\_nr\_of\_lsf}{2} \right\rceil} \left(z - z_Q[i]\right).$$

In case n_nr_of_lsf is odd the polynomials are finally changed to

$$p_Q = p_Q(z+1)(z-1).$$

In case n_nr_of_lsf is even

$$\begin{aligned} p_P &= p_P(z-1) \\ p_Q &= p_Q(z+1) \end{aligned}.$$

The polynomial A(z) is given as

$$A(z) = \frac{p_P(z) + p_Q(z)}{2}.$$

Finally the envelope *H*[*n*], where *n* is the sample index, is calculated as

$$H[n] = \frac{G}{\left| A\left(e^{j\frac{n\pi}{2L+\frac{4S}{3}}} \right) \right|}, \quad n = \left[0{,}2L + \frac{4S}{3} - 1\right].$$

The noise sequence is multiplied by the temporal envelope. Note, that in the case n_nr_of_lsf == 0, *H*[*n*] is defined as

$$H[n] = G, \quad n = \left[0{,}2L + \frac{4S}{3} - 1\right].$$

8.6.3.3 Noise filtering

The structure of the Laguerre synthesis filter, which is applied after the overlap-add operation, is given in Figure 8.17.

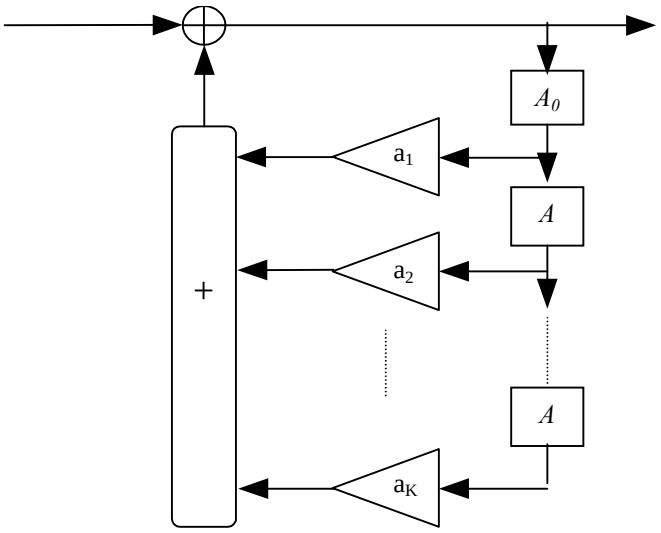


Figure 8.17 — Structure of Laguerre synthesis filter

The filter A_0 is a first-order filter, the filters A are first-order all-pass sections and λ is the so called Laguerre pole whose absolute value is less than 1.

$$A_0(z) = \sqrt{1-\lambda^2}\,\frac{z^{-1}}{1-z^{-1}\lambda}$$
$$A(z) = \frac{-\lambda + z^{-1}}{1-z^{-1}\lambda}$$

Note that a value of 0 for λ is equivalent to conventional LPC.

The parameters for the Laguerre filtering are updated every 2 sub-frames (L samples). In order to make sure that in the generation of the first sample already the desired spectral density is obtained, the initial filter states need to be set. This is realized by copying the final states after the generation of an interval into the initial states for the generation of the next interval. In case that refresh_noise == %0, or start of decoding, the initial filter states will be set to 0. So, using a first set of parameters, two consecutive sub-frames are filtered. For the next 2 sub-frames (with new parameters) the final filter-states resulting from the previous filter operation are used as initial states. For the updates of the LAR parameters differential coding is employed with respect to the LAR parameters of the previous interval.

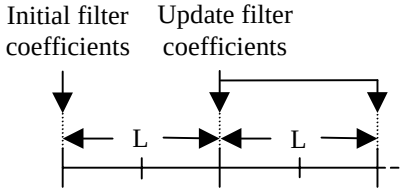


Figure 8.18 — In order to set the initial states of the filter, the final states after the generation of an interval of 2L samples are copied into the initial states for the generation of the next interval of 2L samples

In the bit-stream, for each sub-frame, coefficients for a Laguerre filter structure are encoded in a Log Area Ratio (LAR) notation. The first step of the reconstruction of the Laguerre parameters is the dequantisation of

the LARs (see section 8.6.3.3.1). The parcor coefficients ('rfc') are obtained from the dequantised LARs according to the procedure depicted in section 8.6.3.3.2. The parcors are transformed to FIR coefficients (see section 8.6.3.3.3). The last step is the conversion of FIR coefficients back to the Laguerre coefficients *a* (see section 8.6.3.3.4).

8.6.3.3.1 Quantized LARs

The LAR coefficients in the denominator are de-quantized by multiplying the value encoded in the bit-stream with a constant Δ_{LAR} which is defined as

$$\Delta_{LAR} = \frac{dynr}{levels - 1},$$

where dynr=2*8 is the dynamic range of the LAR coefficients (from –8 to +8), and levels=2^{bits}-2, with bits=9, represents the number of representation levels.

8.6.3.3.2 Convert LARs into parcors

The following algorithm describes the conversion of m LAR coefficients $nlar_q$ into *m* (*m*=n_nrof_den) parcors 'rfc'.

```
for (i=0; i<m; i++)
{
    rfc[i]=(exp(nlar_q[i])-1)/(exp(nlar_q[i])+1)
}
```

8.6.3.3.3 Convert parcors into FIR coefficients

The following algorithm describes the conversion of m parcor coefficients 'rfc' into m+1 a-parameters 'p'.

```
for (k=0; k<m; k++)
{
   d[k] = –rfc[k];
   for (i=0;i<k;i++)
   {
        d[i] = tmp[i]+rfc[k]*tmp[k-i-1];
   }
   for (i=0;i<=k;i++)
   {
        tmp[i] = d[i];
   }
}
p[0] = 1.0;
for (k = 0; k < m; k++)
{
   p[k+1] = -d[k];
}
```

8.6.3.3.4 Convert FIR coefficients into Laguerre coefficients

The a-parameters *p* are converted back to Laguerre coefficients *a* using the following algorithm

$$a'_M = p_M,$$
$$a'_m = p_m - a'_{m+1}\lambda,$$

where $k = [\text{n\_nrof\_den} - 1..0]$ and

$$a_m = -\frac{a_m'}{a_0'}\sqrt{1-\lambda^2}.$$

8.6.4 Parametric stereo

8.6.4.1 Stereo parameters

Three different types of stereo parameters are used in representing the stereo image. For in total a maximum of 34 bands, one set of stereo parameters is available per band.

1) The inter-channel intensity difference, or IID, defined by the relative levels of the band-limited signal.

2) The inter-channel and overall phase differences, IPD and OPD, defining the phase behaviour of the band-limited signal.

3) The inter-channel coherence ICC, defining the (dis)similarity of the left and right band-limited signal.

Figure 8.19 diagrams the processing chain of the parametric stereo decoder.

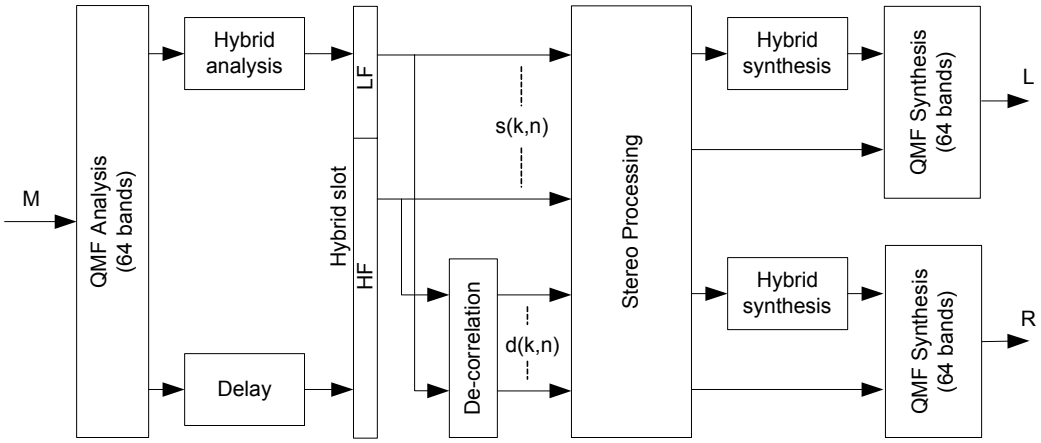


Figure 8.19 — QMF based parametric stereo synthesis

The input to the parametric stereo decoder consists of the monaural parametric generated signal M as obtained through transient, sinusoidal and noise synthesis. The output consists of the left and right stereo representation respectively. In the next paragraphs each block will be detailed.

8.6.4.2 QMF analysis filterbank

This filterbank is identical to the 64 complex QMF analysis filterbank as defined in ISO/IEC 14496-3/AMD1:2003, subclause 4.B.18.2. However, in the equation for matrix M(k,n) and in Figure 4.B.20, the term “(2*n+1)” has to be substituted by “(2*n-1)”. The input to the filterbank are blocks of 64 samples of the monaural synthesized signal M. For each block the filterbank outputs one slot of 64 QMF samples.

8.6.4.3 Low frequency filtering

The lower QMF subbands are further split in order to obtain a higher frequency resolution enabling a proper stereo analysis and synthesis for the lower frequencies. Depending on the number of stereo bands, two hybrid configurations have been defined. See Table 8.36 for an overview of the splits and the type of filter that is used to make the split.

Table 8.36 — Overview of low frequency split for the available configurations.

Configuration, number of stereo bands	QMF subband p	Number of bands Q^p	Filter
10, 20	0	8	Type A
	1	2	Type B
	2	2	
34	0	12	Type A
	1	8	
	2	4	
	3	4	
	4	4	

$$TypeA: G_q^p = g^p[n]\exp(j\frac{2\pi}{Q^p}(q+\frac{1}{2})(n-6)),$$

$$TypeB: G_q^p = g^p[n]\cos(\frac{2\pi}{Q^p}q(n-6)),$$

where g^p represents the prototype filters in QMF subband p. Q^p represents the number of sub-subbands in QMF subband p, q the sub-subband index in QMF channel p and n the time index. The prototype filters are all of length 13 and have a delay of 6 QMF samples. The prototype filters are listed in Table 8.37 and Table 8.38 for the 10,20 and the 34 stereo bands configuration respectively.

Table 8.37 — Prototype filter coefficients for the filters that split the lower QMF subbands for the 10 and 20 stereo bands configuration.

n	$g^0[n]$, Q^0=8	$g^{1,2}[n]$, $Q^{1,2}$=2
0	0.00746082949812	0
1	0.02270420949825	0.01899487526049
2	0.04546865930473	0
3	0.07266113929591	-0.07293139167538
4	0.09885108575264	0
5	0.11793710567217	0.30596630545168
6	0.125	0.5
7	0.11793710567217	0.30596630545168
8	0.09885108575264	0
9	0.07266113929591	-0.07293139167538
10	0.04546865930473	0
11	0.02270420949825	0.01899487526049
12	0.00746082949812	0

Table 8.38 — Prototype filter coefficients for the filters that split the lower QMF subbands for the 34 stereo bands configuration.

n	$g^0[n]$, Q^0 =12	$g^1[n]$, Q^1 =8	$g^{2,3,4}$, $Q^{2,3,4}$ =4
0	0.04081179924692	0.01565675600122	-0.05908211155639
1	0.03812810994926	0.03752716391991	-0.04871498374946
2	0.05144908135699	0.05417891378782	0
3	0.06399831151592	0.08417044116767	0.07778723915851
4	0.07428313801106	0.10307344158036	0.16486303567403
5	0.08100347892914	0.12222452249753	0.23279856662996
6	0.08333333333333	0.12500000000000	0.25000000000000
7	0.08100347892914	0.12222452249753	0.23279856662996
8	0.07428313801106	0.10307344158036	0.16486303567403
9	0.06399831151592	0.08417044116767	0.07778723915851
10	0.05144908135699	0.05417891378782	0
11	0.03812810994926	0.03752716391991	-0.04871498374946
12	0.04081179924692	0.01565675600122	-0.05908211155639

Figure 8.20 and Figure 8.21 illustrate the hybrid analysis and synthesis filterbank for the 10 and 20 stereo bands configuration respectively. Figure 8.22 and Figure 8.23 illustrate the hybrid analysis and synthesis filterbank for the 34 stereo bands configuration respectively. Note that for the 10 and 20 stereo bands configuration, sub-subbands have been combined into a single sub-subband.

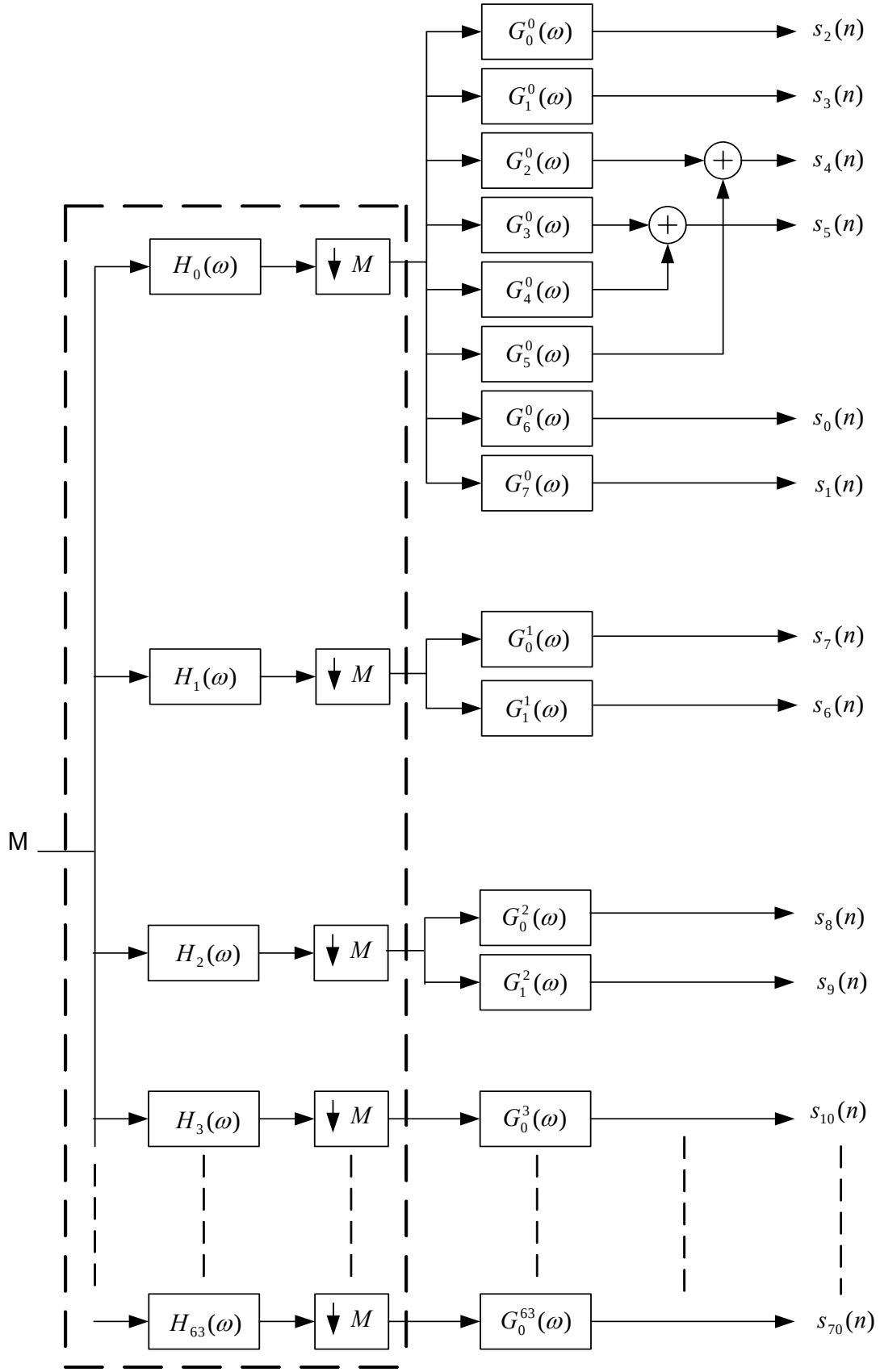


Figure 8.20 — Hybrid QMF analysis filterbank for the 10 and 20 stereo-bands configuration. The lower subbands of the 64 QMF (see dashed box) are further split to provide for increased resolution for the lower frequencies

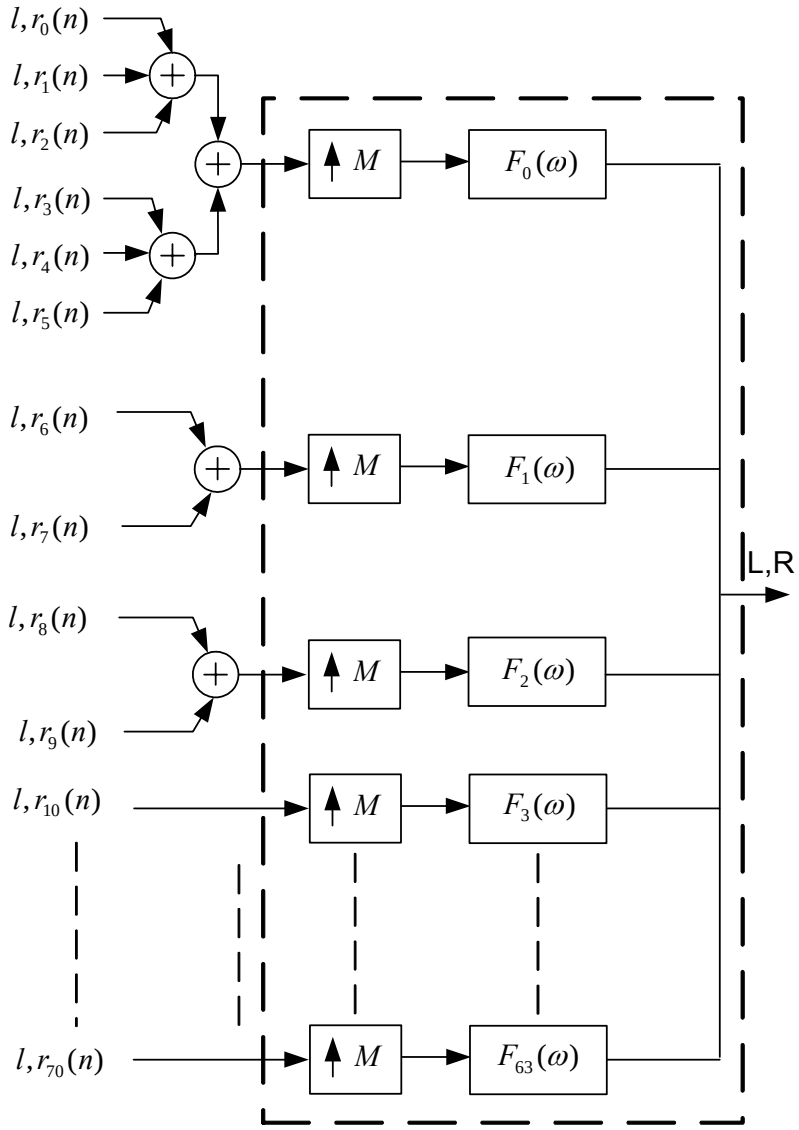


Figure 8.21 — Hybrid QMF synthesis filterbank for the 10 and 20 stereo-bands configuration. The coefficients offering higher resolution for the lower QMF channel are simply added prior to the synthesis with the 64 subbands QMF (see dashed box)

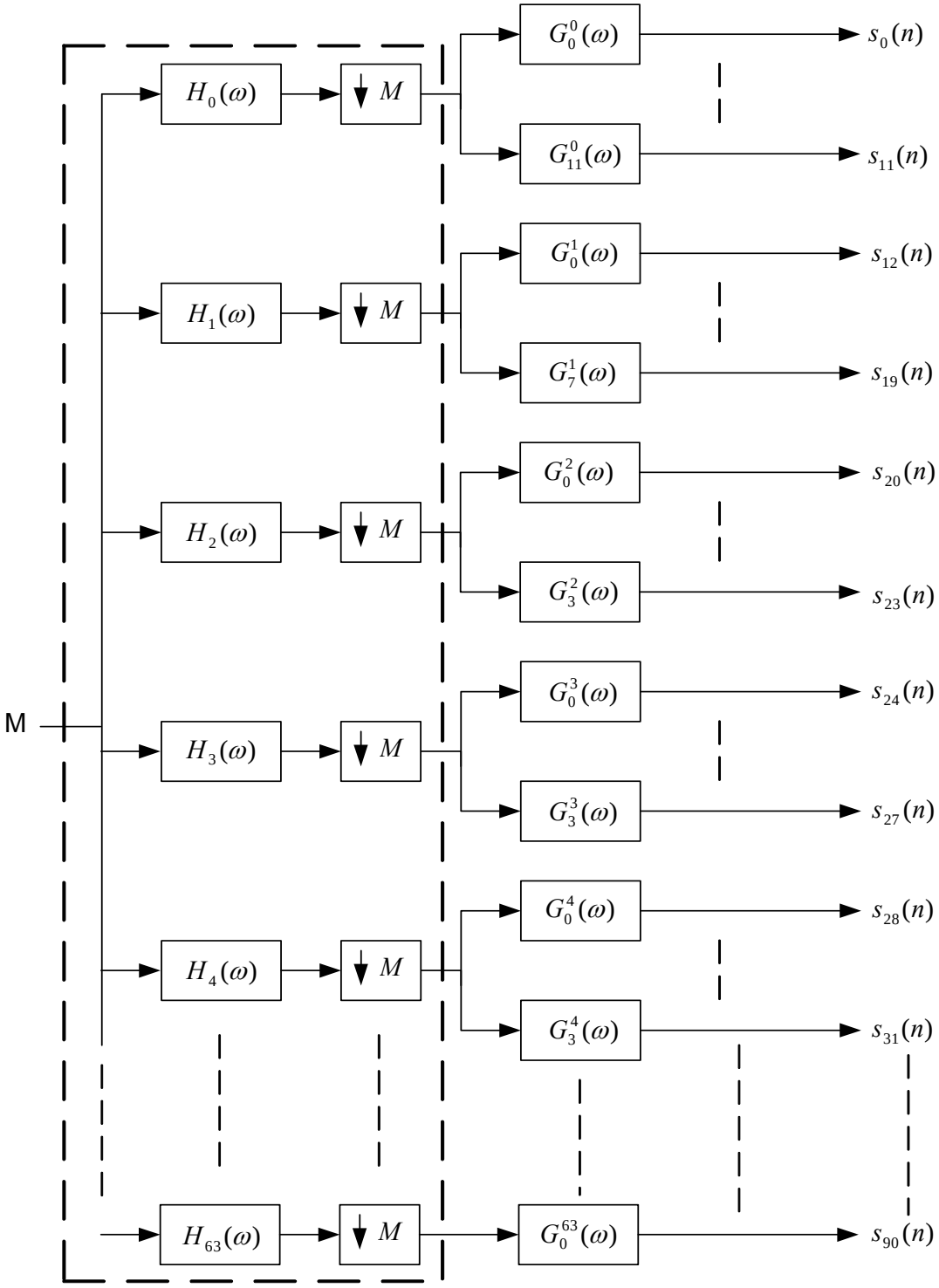


Figure 8.22 — Hybrid QMF analysis filterbank for the 34 stereo-bands configuration. The lower subbands of the 64 QMF (see dashed box) are further split to provide for increased resolution for the lower frequencies

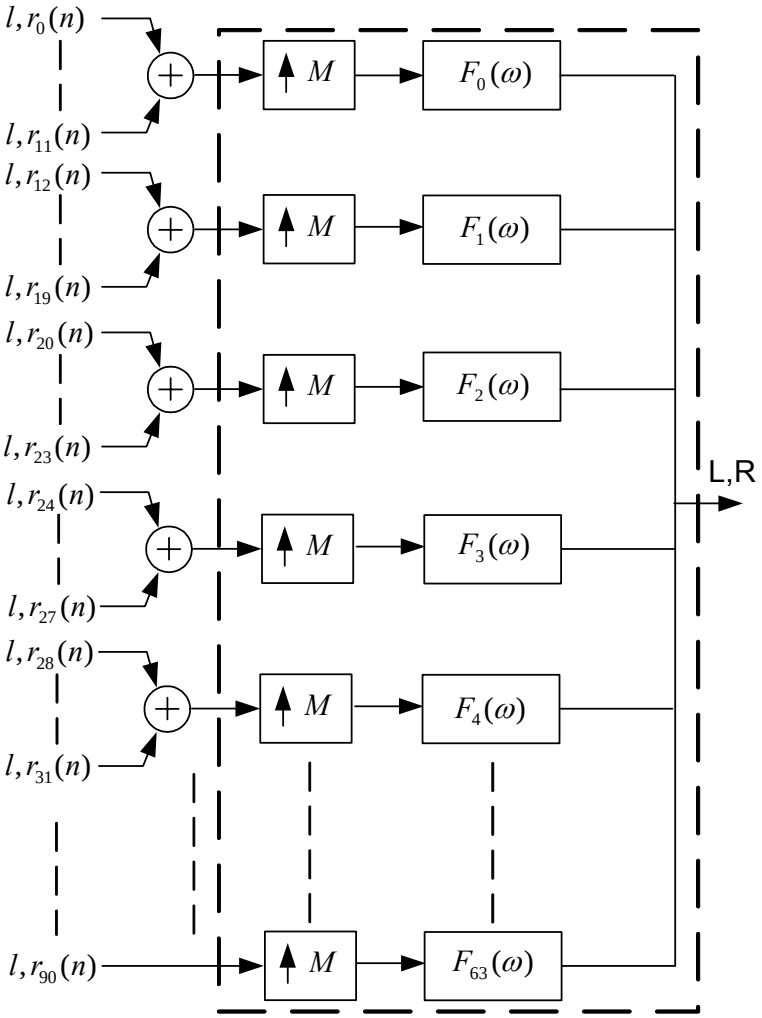


Figure 8.23 — Hybrid QMF synthesis filterbank for the 34 stereo-bands configuration. The coefficients offering higher resolution for the lower QMF subbands are simply added prior to the synthesis with the 64 subbands QMF (see dashed box)

In order to time align all the samples originating from the hybrid filterbank, the remaining QMF subbands that have not been filtered are delay compensated. This delay amounts to 6 QMF subband samples. This means $G_0^k(z) = z^{-6}$ for k=[3...63] (10,20 stereo bands) or k=[5...63] (34 stereo bands). In order to compensate for the overall delay of the hybrid analysis filterbank, the first 10 sets (6 from delay and 4 from QMF filter) of hybrid subbands are flushed and therefore not taken into account for processing. Note that in Figure 8.24 this delay has already been accounted for.

The resultant of this operation is a slot of hybrid subband samples consisting of a LF (low frequency) sub QMF subband portion and HF (high frequency) QMF subband portion. As an illustration, see the hybrid slot in Figure 8.24.

8.6.4.4 Framing

One frame of parametric audio comprises two stereo frames of data. Stereo parameters within a stereo frame can be assigned to one or more slots. The stereo frame boundaries and the positions n_e of the slots that have been assigned stereo parameters to, define so called regions. Stereo parameters are defined for the last slot in a region. By means of these regions transients can be effectively handled. As illustrated by the bold solid lines in Figure 8.24, stereo parameters for non-assigned slots are obtained by means of interpolation. For the very first region ($region_0$), interpolation is performed with respect to the default parameters for index=0 (i.e. IID=0, ICC=1 and IPD=OPD=0). Any existing slots after the last assigned slot in the stereo frame obtain the

same stereo parameters from this last assigned slot ($region_2$). At stereo-frame borders ($region_3$), interpolation for the first region is performed with respect to the last slot in the previous frame. The stereo reconstruction is applied per region. For simplicity, the envelope index is only indicated when applicable.

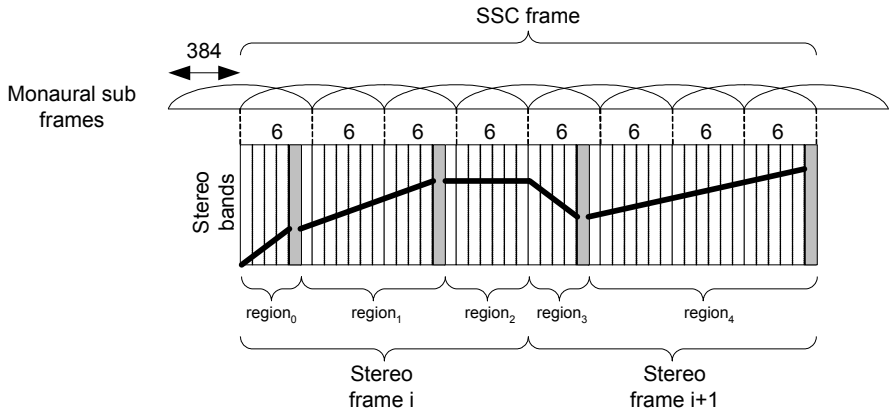


Figure 8.24 — One SSC frame comprises two stereo frames of data. The solid line illustrates the interpolation between stereo parameters for slots that have not been assigned stereo parameters to. Note that the delay introduced by the hybrid QMF analysis filterbank has been compensated for in this figure

8.6.4.5 De-correlation

By means of all-pass filtering and delaying, the sub subband samples $s_k(n)$ are converted into de-correlated sub subband samples $d_k(n)$, where k represents the frequency in the hybrid spectrum and n the time index.

8.6.4.5.1 Constants

$DECAY\_SLOPE = 0.05$ is the all-pass filter decay slope.

$NR\_ALLPASS\_LINKS = 3$ is the number of filter links for the all-pass filter.

NR_PAR_BANDS is the number of frequency bands that can be addressed by the parameter index, b(k) (see Table 8.48 and Table 8.49).

$$NR\_PAR\_BANDS = \begin{cases} 20 & ,10 \text{ or } 20 \text{ stereo bands} \\ 34 & ,34 \text{ stereo bands} \end{cases}$$

NR_BANDS is the number of frequency bands that can be addressed by the sub subband index, k.

$$NR\_BANDS = \begin{cases} 71 & ,10 \text{ or } 20 \text{ stereo bands} \\ 91 & ,34 \text{ stereo bands} \end{cases}$$

$DECAY\_CUTOFF$ is the start frequency band for the all-pass filter decay slope.

$$DECAY\_CUTOFF = \begin{cases} 10 & ,10 \text{ or } 20 \text{ stereo bands} \\ 32 & ,34 \text{ stereo bands} \end{cases}$$

$NR\_ALLPASS\_BANDS$ is the number of all-pass filter bands.

$$NR\_ALLPASS\_BANDS = \begin{cases} 30 & ,10 \text{ or } 20 \text{ stereo bands} \\ 50 & ,34 \text{ stereo bands} \end{cases}$$

$SHORT\_DELAY\_BAND$ is the first stereo band using the short, one sample delay.

$$SHORT\_DELAY\_BAND = \begin{cases} 42 & ,10 \text{ or } 20 \text{ stereo bands} \\ 62 & ,34 \text{ stereo bands} \end{cases}$$

$a_{Smooth} = 0.25$ is the smoothing coefficient.

8.6.4.5.2 Calculate decorrelated signal, $d_k(z)$

The decorrelation process for the first $NR\_ALLPASS\_BANDS$ frequency bands of $s_k(n)$ is based on an all-pass filter described in the Z-domain below. Its transfer function for each band, k is defined by:

$$\mathbf{H}_k(z) = z^{-2} \cdot \varphi_{Fract}(k) \cdot \prod_{m=0}^{NR\_ALLPASS\_LINKS-1} \frac{\mathbf{Q}_{Fract\_allpass}(k,m) z^{-\mathbf{d}(m)} - \mathbf{a}(m)\mathbf{g}_{DecaySlope}(k)}{1 - \mathbf{a}(m)\mathbf{g}_{DecaySlope}(k)\mathbf{Q}_{Fract\_allpass}(k,m) z^{-\mathbf{d}(m)}} \quad ,$$

for $0 \le k < NR\_ALLPASS\_BANDS$.

The fractional delay length matrix, $\mathbf{Q}_{Fract\_allpass}(k,m)$ and the fractional delay vector, $\varphi_{Fract}(k)$ are defined, by:

$$\mathbf{Q}_{Fract\_allpass}(k,m) = \exp\left(-i\pi\mathbf{q}(m)\mathbf{f}_{center}(k)\right) \quad ,\begin{cases} 0 \le k < NR\_ALLPASS\_BANDS \\ 0 \le m < NUM\_OF\_LINKS \end{cases},$$

and

$$\varphi_{Fract}(k) = \exp\left(-i\pi q_{\varphi}\mathbf{f}_{center}(k)\right) \quad ,0 \le k < NR\_ALLPASS\_BANDS\,,$$

where $i = \sqrt{-1}$ denotes the imaginary unit. For the frequency vector, $\mathbf{f}_{center}$, the fractional delay length vector, see Table 8.40 and Table 8.41. The vector $\mathbf{q}(k)$ is defined in Table 8.42. The fractional delay length constant, $q_{\varphi} = 0.39$.

For the filter coefficient vector $\mathbf{a}(m)$ and the delay length vector $\mathbf{d}(m)$ see Table 8.39.

The vector $\mathbf{g}_{DecaySlope}$ contains time invariant factors for making the all-pass filter frequency variant. It is defined by:

$$\mathbf{g}_{DecaySlope}(k) = \begin{cases} \max\langle 0, 1 - DECAY\_SLOPE \cdot (k - DECAY\_CUTOFF) \rangle & , k > DECAY\_CUTOFF \\ 1 & , otherwise \end{cases}$$

for $0 \le k < NR\_ALLPASS\_BANDS$.

For the upper bands, i.e. for $NR\_ALLPASS\_BANDS \le k < NR\_BANDS$ the transfer function, $\mathbf{H}_k(z)$ equals a delay according to:

$\mathbf{H}_k(z) = z^{-D(k)}$, where $D(k)$ is defined by:

$$D(k) = \begin{cases} 14 & , NR\_ALLPASS\_BANDS \le k < SHORT\_DELAY\_BAND \\ 1 & , SHORT\_DELAY\_BAND \le k < NR\_BANDS \end{cases}.$$

8.6.4.5.3 Perform transient detection

To be able to handle transients and other fast time-envelopes, the all-pass filter has to be attenuated at those signals. It is done by the following scheme:

First define the input power matrix, $\mathbf{P}(i,n)$ that contains the sum of the squared sub subband samples of each parameter band. As indicated in Figure 8.25, *n* runs from n_0 to n_{L-1}.

$$\mathbf{P}(i,n) = \sum_{i=b(k)} \left| s_k(n) \right|^2 \quad , 0 \le i < NR\_PAR\_BANDS ,$$

where b(k) is defined in Table 8.48 and Table 8.49. Apply peak decay on the input power signal according to:

$$\mathrm{P}_{PeakDecayNrg}(i,n) = \begin{cases} \mathrm{P}(i,n) & , \alpha \mathrm{P}_{PeakDecayNrg}(i,n-1) < \mathrm{P}(i,n) \\ \alpha \mathrm{P}_{PeakDecayNrg}(i,n-1) & , otherwise \end{cases}.$$

for $0 \le i < NR\_PAR\_BANDS$. α is the peak decay factor defined in Table 8.43.

Subsequently, filter the input power and peak decay power signals with the Z-domain transfer function,

$H_{Smooth}(z)$:

$$\mathbf{P}_{SmoothNrg}(i,z) = H_{Smooth}(z) \cdot \mathbf{P}(i,z),$$

$$\mathbf{P}_{SmoothPeakDecayDiffNrg}(i,z) = H_{Smooth}(z) \cdot \left(\mathbf{P}_{PeakDecayNrg}(i,z) - \mathbf{P}(i,z) \right),$$

for $0 \le i < NR\_PAR\_BANDS$, where

$$H_{Smooth}(z) = \frac{a_{Smooth}}{1 + (a_{Smooth} - 1) \cdot z^{-1}}.$$

The transient attenuator, $\mathbf{G}_{TransientRatio}$ is then calculated as follows:

$$\mathbf{G}_{TransientRatio}(i,n) = \begin{cases} \dfrac{\mathbf{P}_{SmoothNrg}(i,n)}{\gamma \cdot \mathbf{P}_{SmoothPeakDecayDiffNrg}(i,n)} & , \gamma \cdot \mathbf{P}_{SmoothPeakDecayDiffNrg}(i,n) > \mathbf{P}_{SmoothNrg}(i,n) \\ 1 & , otherwise \end{cases},$$

for $0 \le i < NR\_PAR\_BANDS$, where $\gamma = 1.5$ is a transient impact factor.

Finally map the transient attenuator, $\mathbf{G}_{TransientRatio}$ to bands according to:

$$\mathbf{G}_{TransientRatioMapped}(k,n) = \mathbf{G}_{TransientRatio}(b(k),n) \quad , 0 \le k < NR\_BANDS.$$

8.6.4.5.4 Apply transient reduction to decorrelated signal

Let $d_k(z)$ be the decorrelated signal and $s_k(z)$ the mono input signal in the Z-domain for each band. Then $d_k(z)$ is defined according to:

$d_k(z) = \mathbf{G}_{TransientRatioMapped}(k,z) \cdot \mathbf{H}_k(z) \cdot s_k(z)$, where $0 \le k < NR\_BANDS$.

Table 8.39 — Filter coefficient vector, delay length vectors $\mathbf{d}_{24kHz}(m)$ and $\mathbf{d}_{48kHz}(m)$.

m	$\mathbf{a}(m)$	$\mathbf{d}(m)$
0	0.65143905753106	3
1	0.56471812200776	4
2	0.48954165955695	5

Table 8.40 — Delay length vector $\mathbf{f}_{center\_20}$.

k	$\mathbf{f}_{center\_20}(k)$	k	$\mathbf{f}_{center\_20}(k)$
0	-3/8	5	7/8
1	-1/8	6	5/4
2	1/8	7	7/4
3	3/8	8	9/4
4	5/8	9	11/4

Table 8.41 — Delay length vector $\mathbf{f}_{center\_34}$.

k	$\mathbf{f}_{center\_34}(k)$	k	$\mathbf{f}_{center\_34}(k)$
0	1/12	16	9/8
1	3/12	17	11/8
2	5/12	18	13/8
3	7/12	19	15/8
4	9/12	20	9/4
5	11/12	21	11/4
6	13/12	22	13/4
7	15/12	23	7/4
8	17/12	24	17/4
9	-5/12	25	11/4
10	-3/12	26	13/4
11	-1/12	27	15/4
12	17/8	28	17/4
13	19/8	29	19/4
14	5/8	30	21/4
15	7/8	31	15/4

$$\mathbf{f}_{center\_20}(k) = k + \frac{1}{2} - 7 \quad ,10 \le k < NR\_ALLPASS\_BANDS\ ,$$

$$\mathbf{f}_{center\_34}(k) = k + \frac{1}{2} - 27 \quad ,32 \le k < NR\_ALLPASS\_BANDS\ ,$$

$$\mathbf{f}_{center} = \begin{cases} \mathbf{f}_{center\_20} & ,NR\_PAR\_BANDS = 20 \\ \mathbf{f}_{center\_34} & ,NR\_PAR\_BANDS = 34 \end{cases} .$$

Table 8.42 — Fractional delay length vector $\mathbf{q}(m)$

m	$\mathbf{q}(m)$
0	0.43
1	0.75
2	0.347

Table 8.43 — Peak Decay Factors α

α	0.76592833836465

8.6.4.6 Stereo Processing

The sets of sub subband samples $s_k(n)$ and $d_k(n)$ are now processed according to the stereo cues. These cues are defined per stereo band. All the hybrid subband samples within a stereo band are processed according to the cues in that respective stereo band. Table 8.48 and Table 8.49 indicate the hybrid subband samples that fall into each stereoband for the (10,20) and 34 stereoband configuration. k traverses the range from [0…70] or [0…90] for the (10,20) or 34 stereoband configuration respectively.

8.6.4.6.1 Mapping

The number of stereo bands that is actually used for the processing of the cues depends on the number of available parameters for IID and ICC according to the relation given in Table 8.44. In case no IID or no ICC parameters have been transmitted in the current frame (enable_iid==%0 or enable_icc==%0), the number of IID or ICC parameters, respectively, is assumed to be 20 for the purpose of Table 8.44. In case no IID and no ICC parameters have been transmitted in the current frame (enable_iid==%0 and enable_icc==%0), the number of stereo bands in the previous frame is kept unchanged and used also for the processing of the current frame.

Table 8.44 — The number of stereo bands depends on the number of parameters for IID and ICC

Number of IID parameters	number of ICC parameters	number of stereo bands
10	10	20 (i.e., 10,20 stereo band configuration)
10	20	
20	10	
20	20	
10,20	34	34
34	10,20	

In the case the number of parameters for IID and ICC differs from the number of stereo bands, as dictated in Table 8.44, a mapping from the lower number to the higher number of parameters is required. For the mapping from 10 to 20 parameters this is realised by duplicating every parameter as shown in Table 8.45. For the mapping from 20 to 34 parameters this is realized according to Table 8.45. For the mapping from 10 to 34 parameters, the 10 parameters are first mapped to 20 parameters and subsequently to 34 parameters. Table 8.46 provides the inverse mapping from 34 to 20 parameters.

Table 8.45 — Mapping from 10 to 20 to 34 parameters

parameter grid			parameter grid		
34	20	10	34	20	10
idx_0	idx_0	idx_0	idx_{17}	idx_{11}	idx_5
idx_1	$(idx_0+idx_1)/2$	idx_0	idx_{18}	idx_{12}	idx_6
idx_2	idx_1	idx_0	idx_{19}	idx_{13}	idx_6
idx_3	idx_2	idx_1	idx_{20}	idx_{14}	idx_7
idx_4	$(idx_2+idx_3)/2$	idx_1	idx_{21}	idx_{14}	idx_7
idx_5	idx_3	idx_1	idx_{22}	idx_{15}	idx_7
idx_6	idx_4	idx_2	idx_{23}	idx_{15}	idx_7
idx_7	idx_4	idx_2	idx_{24}	idx_{16}	idx_8
idx_8	idx_5	idx_2	idx_{25}	idx_{16}	idx_8
idx_9	idx_5	idx_2	idx_{26}	idx_{17}	idx_8
idx_{10}	idx_6	idx_3	idx_{27}	idx_{17}	idx_8
idx_{11}	idx_7	idx_3	idx_{28}	idx_{18}	idx_9
idx_{12}	idx_8	idx_4	idx_{29}	idx_{18}	idx_9
idx_{13}	idx_8	idx_4	idx_{30}	idx_{18}	idx_9
idx_{14}	idx_9	idx_4	idx_{31}	idx_{18}	idx_9
idx_{15}	idx_9	idx_4	idx_{32}	idx_{19}	idx_9
idx_{16}	idx_{10}	idx_5	idx_{33}	idx_{19}	idx_9

Table 8.46 — Mapping of IID, ICC, IPD and OPD parameters from 34 stereo bands to 20 stereo bands. For IPD and OPD parameters, this mapping applies up to and including idx_{10} and idx_{16} for 20 and 34 stereo bands respectively

20 stereo bands	34 stereo bands
idx_0	$(2*idx_0+idx_1)/3$
idx_1	$(idx_1+2*idx_2)/3$
idx_2	$(2*idx_3+idx_4)/3$
idx_3	$(idx_4+2*idx_5)/3$
idx_4	$(idx_6+idx_7)/2$
idx_5	$(idx_8+idx_9)/2$
idx_6	idx_{10}
idx_7	idx_{11}
idx_8	$(idx_{12}+idx_{13})/2$
idx_9	$(idx_{14}+idx_{15})/2$
idx_{10}	idx_{16}
idx_{11}	idx_{17}
idx_{12}	idx_{18}
idx_{13}	idx_{19}
idx_{14}	$(idx_{20}+idx_{21})/2$
idx_{15}	$(idx_{22}+idx_{23})/2$
idx_{16}	$(idx_{24}+idx_{25})/2$
idx_{17}	$(idx_{26}+idx_{27})/2$
idx_{18}	$(idx_{28}+idx_{29}+idx_{30}+idx_{31})/4$
idx_{19}	$(idx_{32}+idx_{33})/2$

The averaging process denoted by e.g. $(2*idx_0 + idx_1)/2$ in Table 8.45 and Table 8.46 is carried out for the integer index representation idx_k of the IID or ICC parameters prior to dequantization, according to ANSI-C integer arithmetic.

The IPD/OPD parameters follow the mapping for the IID parameters, taking into account the relative amount of parameters for IPD/OPD as given by Table 8.24. Consequently the same mapping as for IID is applied, but only for a lower number of parameters for IPD/OPD. For the upper stereo bands, where no IPD/OPD data is transmitted, the IPD/OPD parameters are set to zero.

If the number of stereo bands changes from 10,20 in the previous frame to 34 in the current frame, the coefficients $h_{11}(b)$, $h_{12}(b)$, $h_{21}(b)$, and $h_{22}(b)$ at the end of the previous frame are mapped from 20 to 34 stereo bands according to Table 8.40 (by substituting idx_b by $h_{ij}(b)$, where ij is 11, 12, 21, or 22) prior to further processing as defined in subclause 8.6.4.6.3 and the IPD/OPD smoothing state variables are reset, i.e., $opd(b, n_{e-1}) = 0$, $ipd(b, n_{e-1}) = 0$, $opd(b, n_e) = 0$, and $ipd(b, n_e) = 0$. The frequency resolution of the hybrid QMF analysis filterbank (see subclause 8.6.4.3) is changed instantaneously to the 34 stereo band configuration. The state variable of the decorrelation process are reset to zero (see Table 8.47).

If the number of stereo bands changes from 34 in the previous frame to 10,20 in the current frame, the coefficients $h_{11}(b)$, $h_{12}(b)$, $h_{21}(b)$, and $h_{22}(b)$ at the end of the previous frame are mapped from 34 to 20 stereo bands according to Table 8.46 (by substituting idx_b by $h_{ij}(b)$, where ij is 11, 12, 21, or 22) prior to further processing as defined in subclause 8.6.4.6.3 and the IPD/OPD smoothing state variables are reset, i.e., $opd(b, n_{e-1}) = 0$, $ipd(b, n_{e-1}) = 0$, $opd(b, n_e) = 0$, and $ipd(b, n_e) = 0$. The frequency resolution of the hybrid QMF analysis filterbank (see subclause 8.6.4.3) is changed instantaneously to the 20 stereo band configuration. The state variable of the decorrelation process are reset to zero (see Table 8.47).

Table 8.47 — Changing number of stereo bands.

	current frame	
previous frame	10/20 bands	34 bands
10/20 bands	-	map $h_{ij}(b)$ according to Table 8.45, reset state variables
34 bands	map $h_{ij}(b)$ according to Table 8.46, reset state variables	-

8.6.4.6.2 Mixing

In order to generate the QMF subband signals for the subband samples $n = n_e + 1...n_{e+1}$ the parameters at position n_e and n_{e+1} are required as well as the subband domain signals $s_k(n)$ and $d_k(n)$ for $n = n_e + 1...n_{e+1}$ (see Figure 8.25). For IPD/OPD, the parameters at position n_{e-1} are needed in addition. n_e represents the start position for envelope e. In the case frameclass == %1 (VAR_BORDERS), the border positions n_e are obtained by borderposition[e]. In the case frameclass == %0 (FIX_BORDERS), the border positions n_e are obtained by means of

$$n_e = \left\lfloor \frac{numQMFSlots * (e+1)}{num\_env} \right\rfloor - 1, \qquad e = [0,...,num\_env - 1].$$

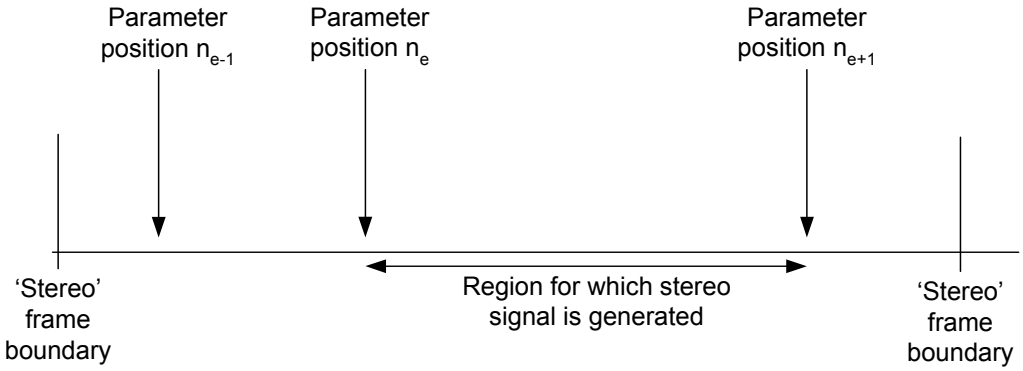


Figure 8.25 — Region for which the stereo subband signals are generated

The stereo sub subband signals are constructed as:

$$l_k(n) = H_{11}(k,n)s_k(n) + H_{21}(k,n)d_k(n)$$
$$r_k(n) = H_{12}(k,n)s_k(n) + H_{22}(k,n)d_k(n).$$

In order to obtain the matrices $H_{11}(k,n)$, $H_{12}(k,n)$, $H_{21}(k,n)$, $H_{22}(k,n)$, the vectors $h_{11}(b)$, $h_{12}(b)$, $h_{21}(b)$, $h_{22}(b)$ need to be calculated first, where the parameter *b* is used as parameter index. First for the parameter position n_{e+1} the intensity differences (IID) are transformed to the linear domain.

$$c(b) = 10^{\frac{iid(b)}{20}},$$

where *iid*(*b*) represents the decoded IID value for stereo band *b* in dB. Depending on the ICC mode configuration, either mixing procedure R_a or R_b is used, see Table 8.27. For both mixing procedures, the parameters for parameter position n_{e+1} are used.

8.6.4.6.2.1 Mixing procedure R_a

In the case mixing procedure R_a is used the following method is applied.

From the intensity differences two scale-factor vectors c_1 and c_2 are calculated.

$$c_1(b) = \frac{\sqrt{2}}{\sqrt{1 + c^2(b)}}$$
$$c_2(b) = \frac{\sqrt{2}c(b)}{\sqrt{1 + c^2(b)}}$$

From these and the ICC parameter $\rho(b)$, the coefficients $h_{xy}(b)$ are calculated according to

$$\alpha(b) = \frac{1}{2}\arccos(\rho(b))$$
$$\beta(b) = \alpha(b)\frac{c_1(b) - c_2(b)}{\sqrt{2}}$$
$$h_{11}(b) = \cos(\alpha(b) + \beta(b))c_2(b)$$
$$h_{12}(b) = \cos(\beta(b) - \alpha(b))c_1(b)$$
$$h_{21}(b) = \sin(\alpha(b) + \beta(b))c_2(b)$$
$$h_{22}(b) = \sin(\beta(b) - \alpha(b))c_1(b)$$

8.6.4.6.2.2 Mixing procedure R_b

In the case mixing procedure R_b is used the following method is applied.

In order to prevent instabilities, in the case the value of $\rho(b)$ is smaller than 0.05, $\rho(b)$ is set to 0.05. In the case $c(b)$ is not equal to 1

$$\alpha(b) = \frac{1}{2}\arctan\left(\frac{2c(b)\rho(b)}{c^2(b) - 1}\right),$$

otherwise $\alpha(b) = \frac{\pi}{4}$. After modulo correction of $\alpha(b)$, again the value of $c(b)$ and $\rho(b)$ are used to derive the coefficients $h_{xy}(b)$.

$$\alpha(b) = \alpha(b) - \left\lfloor \frac{\alpha(b)}{\frac{1}{2}\pi} \right\rfloor \frac{1}{2}\pi,$$

$$\mu(b) = 1 + \frac{4\rho^2(b) - 4}{(c(b) + c^{-1}(b))^2},$$

$$\gamma(b) = \arctan\left(\sqrt{\frac{1-\sqrt{\mu(b)}}{1+\sqrt{\mu(b)}}}\right),$$

$$h_{11}(b) = \sqrt{2}\cos(\alpha(b))\cos(\gamma(b)), \ .$$

$$h_{12}(b) = \sqrt{2}\sin(\alpha(b))\cos(\gamma(b)),$$

$$h_{21}(b) = -\sqrt{2}\sin(\alpha(b))\sin(\gamma(b)),$$

$$h_{22}(b) = \sqrt{2}\cos(\alpha(b))\sin(\gamma(b)).$$

8.6.4.6.3 Phase parameters

8.6.4.6.3.1 Phase parameters disabled

In the case IPD and OPD are disabled as indicated by (**enable_ipdopd==**0) the following procedure is applied. In order to obtain $H_{11}(k, n_{e+1})$, $H_{12}(k, n_{e+1})$, $H_{21}(k, n_{e+1})$ and $H_{22}(k, n_{e+1})$ we use the following equations

$$\begin{aligned} H_{11}(k, n_{e+1}) &= h_{11}(b(k)) \\ H_{12}(k, n_{e+1}) &= h_{12}(b(k)) \\ H_{21}(k, n_{e+1}) &= h_{21}(b(k)) \\ H_{22}(k, n_{e+1}) &= h_{22}(b(k)) \end{aligned},$$

where *b*(*k*) is defined in Table 8.48 and Table 8.49.

8.6.4.6.3.2 Phase parameters enabled

In the case IPD and OPD are enabled as indicated by (enable_ipdopd==1) the following procedure is applied. First the IPD and OPD values are smoothed over time according to

$$\varphi_{opd}(b) = \angle\left\{\frac{1}{4}\exp(j \cdot opd(b, n_{e-1})) + \frac{1}{2}\exp(j \cdot opd(b, n_e)) + \exp(j \cdot opd(b, n_{e+1}))\right\}$$

$$\varphi_{ipd}(b) = \angle\left\{\frac{1}{4}\exp(j \cdot ipd(b, n_{e-1})) + \frac{1}{2}\exp(j \cdot ipd(b, n_e)) + \exp(j \cdot ipd(b, n_{e+1}))\right\} .$$

In the case the number of IPD/OPD parameters for parameter position n_{e-1} and/or n_e are different from the number of IPD/OPD parameters for parameter position n_{e+1}, these are mapped to the number of IPD/OPD parameters for parameter position n_{e+1} using Table 8.45 and Table 8.46.

$$\begin{aligned} \varphi_1(b) &= \varphi_{opd}(b) \\ \varphi_2(b) &= \varphi_{opd}(b) - \varphi_{ipd}(b) \end{aligned}.$$

Finally, in order to get to $H_{11}(k, n_{e+1})$, $H_{12}(k, n_{e+1})$, $H_{21}(k, n_{e+1})$ and $H_{22}(k, n_{e+1})$, the following equations are applied.

$$\begin{aligned} H_{11}(k, n_{e+1}) &= h_{11}(b(k)) \cdot \exp(j\varphi_1(b(k))) \\ H_{12}(k, n_{e+1}) &= h_{12}(b(k)) \cdot \exp(j\varphi_2(b(k))) \\ H_{21}(k, n_{e+1}) &= h_{21}(b(k)) \cdot \exp(j\varphi_1(b(k))) \\ H_{22}(k, n_{e+1}) &= h_{22}(b(k)) \cdot \exp(j\varphi_2(b(k))) \end{aligned},$$

where Table 8.48 and Table 8.49 are used to translate from parameter indexing to subband indexing. For indices denoted with a *, we use:

$$\begin{aligned} H_{11}(k, n_{e+1}) &= h_{11}(b(k)) \cdot \exp(-j\varphi_1(b(k))) \\ H_{12}(k, n_{e+1}) &= h_{12}(b(k)) \cdot \exp(-j\varphi_2(b(k))) \\ H_{21}(k, n_{e+1}) &= h_{21}(b(k)) \cdot \exp(-j\varphi_1(b(k))) \\ H_{22}(k, n_{e+1}) &= h_{22}(b(k)) \cdot \exp(-j\varphi_2(b(k))) \end{aligned}.$$

Table 8.48 — Mapping of parameters from 20 bands to 71 sub subbands

sub subband index *k*	QMF channel	Parameter index *b*(*k*)	
0	0	1*	Sub QMF
1	0	0*	
2	0	0	
3	0	1	
4	0	2	
5	0	3	
6	1	4	
7	1	5	
8	2	6	
9	2	7	
10	3	8	QMF (only)
11	4	9	
12	5	10	
13	6	11	
14	7	12	
15	8	13	
16-17	9-10	14	
18-20	11-13	15	
21-24	14-17	16	
25-29	18-22	17	
30-41	23-34	18	
42-70	35-63	19	

Table 8.49 — Mapping of parameters from 34 bands to 91 sub subbands

Sub subband index *k*	QMF channel	Parameter index *b*(*k*)	
0	0	0	Sub QMF
1	0	1	
2	0	2	
3	0	3	
4	0	4	
5	0	5	
6-7	0	6	
8	0	7	
9	0	2*	
10	0	1*	
11	0	0*	
12-13	1	10	
14	1	4	
15	1	5	
16	1	6	
17	1	7	
18	1	8	
19	1	9	
20	2	10	
21	2	11	
22	2	12	
23	2	9	
24	3	14	
25	3	11	
26	3	12	
27	3	13	
28	4	14	
29	4	15	
30	4	16	
31	4	13	
32	5	16	QMF (only)
33	6	17	
34	7	18	
35	8	19	
36	9	20	
37	10	21	
38-39	11-12	22	
40-41	13-14	23	
42-43	15-16	24	
44-45	17-18	25	
46-47	19-20	26	
48-50	21-23	27	
51-53	24-26	28	
54-56	27-29	29	
57-59	30-32	30	
60-63	33-36	31	
64-67	37-40	32	
68-90	41-63	33	

8.6.4.6.4 Interpolation

The intermediate values for $H_{11}(k,n)$, $H_{12}(k,n)$, $H_{21}(k,n)$ and $H_{22}(k,n)$ at positions $n = n_e + 1 ... n_{e+1}$ are obtained by means of linear interpolation conforming to.

$$H_{11}(k,n) = H_{11}(k,n_e) + (n-n_e)\frac{H_{11}(k,n_{e+1}) - H_{11}(k,n_e)}{n_{e+1}-n_e}$$
$$H_{12}(k,n) = H_{12}(k,n_e) + (n-n_e)\frac{H_{12}(k,n_{e+1}) - H_{12}(k,n_e)}{n_{e+1}-n_e}$$
$$H_{21}(k,n) = H_{21}(k,n_e) + (n-n_e)\frac{H_{21}(k,n_{e+1}) - H_{21}(k,n_e)}{n_{e+1}-n_e}$$
$$H_{22}(k,n) = H_{22}(k,n_e) + (n-n_e)\frac{H_{22}(k,n_{e+1}) - H_{22}(k,n_e)}{n_{e+1}-n_e}$$

8.6.4.6.5 Procedure for incomplete parameter sets

In the case no parameters have been transmitted in the current frame for either IID, ICC, nor IPD/OPD or a combination thereof, the parameters values for the current frame are obtained according to the num_env variable as given in, Table 8.50,

Table 8.51 and Table 8.52.

Table 8.50 — Derivation of parameters for IID in the case no parameters are transmitted

	enable_iid	
	0	1
num_env=0	IID parameters set to default	IID parameters held
num_env>0	IID parameters set to default	n.a.

Table 8.51 — Derivation of parameters for ICC in the case no parameters are transmitted

	enable_icc	
	0	1
num_env=0	ICC parameters set to default	ICC parameters held
num_env>0	ICC parameters set to default	n.a.

Table 8.52 — Derivation of parameters for IPD/OPD in the case no parameters are transmitted

	enable_ipdopd	
	0	1
num_env=0	IPD/OPD parameters set to default	IPD/OPD parameters held
num_env>0	IPD/OPD parameters set to default	n.a.

In the case parameters are to be set to default, the parameters at the positions defined by n_e are set to their default value (index=0).

In the case the parameters are to be held, two situations are distinguished. If enable_ipdopd==%1, the four vectors $H_{11}(k,n)$, $H_{12}(k,n)$, $H_{21}(k,n)$ and $H_{22}(k,n)$ for all n=[0,…, *numQMFSlots*-1], are copied from those same four vectors at position n=*numQMFSlots*-1 in the previous ps_data() element. If enable_ipdopd==%0, the four vectors $H_{11}(k,n)$, $H_{12}(k,n)$, $H_{21}(k,n)$ and $H_{22}(k,n)$ for all n=[0,…, *numQMFSlots*-1], are set to the four vectors $h_{11}(k,n)$, $h_{12}(k,n)$, $h_{21}(k,n)$ and $h_{22}(k,n)$ respectively, where n=*numQMFSlots*-1 in the previous ps_data() element.

8.6.4.7 Hybrid QMF synthesis filterbank

The stereo processed hybrid subband signals $l_k(n)$ and $r_k(n)$ are fed into the hybrid synthesis filterbanks, which are implemented as adders of sub QMF samples. This is illustrated in Figure 8.21 and Figure 8.23. The two synthesis filterbanks are identical to the 64 complex QMF synthesis filterbank as defined in ISO/IEC 14496-3/AMD1:2003 subclause 4.6.18.4.2. The input to the filterbank are slots of 64 QMF samples. For each slot the filterbank outputs one block of 64 samples of the one channel of the reconstructed stereo signal. There are two independent instances of the QMF synthesis filterbank for the left and right channel, respectively.

8.6.5 Start/stop situations for decoding

Decoding of an excerpt has to be started and ended in a certain way. This section explains how to deal with start and end of the decoding process.

8.6.5.1 Start decoding

The start of decoding occurs for the first frame of an excerpt, or during random access in an excerpt.

No special actions need to be taken for transients.

For sinusoids a previous (non-existent) sub-frame must be filled with a zero signal. The overlap-add method then generates a natural fade-in for the sinusoidal components of the first sub-frame.

For noise a previous (non-existent) sub-frame must be filled with a zero signal. The overlap-add method then generates a natural fade-in for the noise component of the first sub-frame.

A conformant decoder that receives PS data shall output the mono signal in the two output channels until a first ps_data() element with enable_ps_header==1 is received and in which for all enabled parameters frequency differential coding is employed and num_env>0, ensuring that the PS data can be decoded correctly.

8.6.5.2 Stop decoding

The stop of decoding occurs for the last frame of an excerpt, or during random access in an excerpt (stop decoding process “manually” (e.g. stop, skip, pause)).

For a step transient no special precautions are required. For a Meixner transient it is possible that the tail has not ended at the end of the excerpt. It is advised to stop generating output for the Meixner transient at the end of the excerpt.

For sinusoids a next (non-existent) sub-frame must be filled with a zero signal. The overlap-add method then generates a natural fade-out for the sinusoidal components of the last sub-frame.

For noise a next (non-existing) sub-frame must be filled with a zero signal. The overlap-add method then generates a natural fade-out for the noise component of the last sub-frame.

For parametric stereo no special actions are required.

8.7 References

[FFT] *High-Precision Fourier Analysis of Sounds Using Signal Derivatives.* M. Desainte-Catherine and S. Marchand, Journal of the Audio Engineering Society, Vol. 48, No. 7/8, 2000 July/August.

[Laguerre1] A.C. den Brinker. Stability of linear predictive structures using IIR filters. In *Proc. 12th ProRISC* Workshop, pages 317–320, Veldhoven (NL), 29-30 Nov. 2001.

[Laguerre2] V. Voitishchuk, A.C. den Brinker, and S.J.L. van Eijndhoven. Alternatives for warped linear predictors. In *Proc. 12th ProRISC Worksho*p, pages 710–713, Veld-hoven (NL), 29-30 Nov. 2001.

[HILN] ISO/IEC 14496-3 Version 2 HILN reference code.

[Rothweiler] *A Rootfinding algorithm for line spectral frequencies*. J. Rothweiler. ICASSP March 1991.

[Breebaart] Binaural processing model based on contralateral inhibition I. Model setup, J. Breebaart, S. van de Par and A. Kohlrausch, J. Acoust. Soc. Am., 110:1074–1088, 2001.

Annex 8.A
(normative)

Combination of the SBR tool with the parametric stereo tool

8.A.1 Overview

The parametric stereo coding tool (PS tool) can be used in combination with the SBR tool as defined in subclause 4.6.18. In this case, a 1-channel audio signal is conveyed by AAC+SBR (i.e., HE-AAC) and the PS tool is used to reconstruct a 2-channel stereo signal from this monaural signal. The bitstream element ps_data() as defined in subclause 8.4.2 conveys the information needed by the PS tool and is carried in the sbr_extension() container of the SBR bitstream.

The usage of this parametric stereo extension to HE-AAC is signalled implicitly in the bitstream. Hence, if an sbr_extension() with bs_extension_id==EXTENSION_ID_PS is found in the SBR part of the bitstream, a decoder supporting the combination of SBR and PS shall operate the PS tool to generate a stereo output signal. If no ps_data() element is available in the SBR part of a monaural HE-AAC bitstream, the normal monaural signal is generated by the SBR tool and mapped to a stereo output signal in which the left and right channel both contain the same monaural signal.

8.A.2 Bitstream syntax and semantics

The bitstream element ps_data() as defined in subclause 8.4.2 is carried in the sbr_extension() container (see Table 8.A.1 below) provided by the SBR bitstream defined in subclause 4.4.2.8. The semantics of the bs_extension_id field are given in Table 8.A.2, which replaces Table 4.97 “bs_extension_id.”

Table 8.A.1 — Syntax of sbr_extension()

Syntax	No. of bits	Mnemonic

sbr_extension(bs_extension_id, num_bits_left)
{
  switch (bs_extension_id) {
  case EXTENSION_ID_PS:
  num_bits_left -= ps_data();
  break;
  default:
  bs_fill_bits;
  num_bits_left = 0;
  break;
  }
}
	
	Note 1
	
	num_bits_left
		
		bslbf

Note 1: ps_data() returns the number of bits read.		

Table 8.A.2 — Values of the bs_extension_id field

Symbol	Value	Purpose
EXTENSION_ID_PS	2	Parametric Stereo Coding
	all other values	reserved

8.A.3 Decoding process

Semantics and decoding process for the PS tool are defined in subclauses 8.5.2 and 8.6.4, respectively. When the PS tool is combined with SBR, a stereo frame is identical to an SBR frame and consists of 32

complex samples per QMF band for 1024 framing of the AAC (30 samples for 960 framing). The initial 64-band QMF analysis filterbank of the PS tool is removed and the 64-band QMF representation of the monaural signal generated by the SBR tool as available directly prior to SBR's 64-band QMF synthesis filterbank is used as input to the PS tool. The monaural 64-band QMF synthesis filterbank of the SBR tool is obsolete and hence removed. Instead, the two 64-band QMF synthesis filterbanks at the output of the PS tool are used to generate the stereo audio signal.

As shown in Figure 4.46, "Synchronization and timing" in the SBR tool description, there are 6 QMF samples look-ahead available in the low-band buffer, i.e., $\mathbf{X}_{Low}(k,l)$ with 34<=*l*<40 for 1024 framing. Hence, the hybrid filter structure of the PS tool, where the lowest 3 or 5 QMF bands (all of which are in the low-band) are split up further with help of 13-tap linear-phase FIR filters, does not introduce any algorithmic delay when the PS tool is inserted between SBR processing and the final 64-band QMF synthesis filterbanks. This means that the delay as depicted in Figure 8.19 is obsolete and thus removed.

Hence, the input to the PS tool is a matrix $\mathbf{X}_{input}(k,l)$ defined according to:

$$\mathbf{X}_{input}(k,l) = \begin{cases} \mathbf{X}_{Low}\left(k, l + t_{HFAdj}\right) & , 0 \le k < 5, numTimeSlots \cdot RATE \le l < numTimeSlots \cdot RATE + 6 \\ \mathbf{X}(k,l) & , 0 \le k < 64, 0 \le l < numTimeSlots \cdot RATE \end{cases}$$

where $\mathbf{X}_{Low}(k,l)$, $\mathbf{X}(k,l)$, t_{HFAdj} , *numTimeSlots*, and *RATE* are defined in subclause 4.6.18 SBR Tool. Furthermore, *numQMFSlots=numTimeSlots*RATE.*

In order to allow an efficient implementation of the PS tool, a partial reset of the decorrelator state variables is performed for each stereo frame for all QMF subbands above the highest QMF subband generated by the SBR tool by forcing the states

$$\begin{aligned} d_k(n) &= 0 \\ s_k(n) &= 0 \end{aligned},$$

where $n < n_e$, $k_{\max} \le k < NR\_BANDS$, n_e is the first sample in the current stereo frame and

$$k_{\max} = k_x + M + \begin{cases} 7 & , 10 \text{ or } 20 \text{ stereo bands} \\ 27 & , 34 \text{ stereo bands} \end{cases}$$

where k_x and M are defined in subclause 4.6.18.3.2.2.

The PS tool uses a complex-valued QMF representation and therefore cannot be used in combination with the low power version of the SBR tool. If DRC is used in combination with SBR as defined in subclause 4.5.2.7.5, DRC is applied in the QMF domain to the output of the PS tool immediately prior to the QMF synthesis filterbanks. The same *factor(k,l)* is applied to both the left and the right audio channel.

8.A.4 Baseline version of the parametric stereo coding tool

In order to facilitate implementation of the PS decoder tool on platforms with very limited computational resources, a baseline version of the PS tool is defined. A PS decoder implementing this baseline version always uses the hybrid filter structure for 20 stereo bands and does not implement IPD/OPD synthesis and mixing mode Rb. This results in a reduction of the computational complexity by approximately 25% when compared to unrestricted PS tool. The baseline version of the PS tool supports the complete bitstream syntax for ps_data(). However, IPD/OPD data is ignored and reset to IPD=OPD=0 prior to stereo synthesis. If a 34 stereo band configuration is used for IID or ICC parameters in the bitstream, the decoded parameters are mapped to 20 stereo bands according to Table 8.46. The averaging process denoted by e.g. $(2*idx_0+idx_1)/3$ in this table is carried out according to ANSI-C integer arithmetic for the integer index representation idx_k of the IID or ICC parameters prior to dequantization. The baseline decoder always uses mixing mode Ra, independent from the value of icc_mode.

Annex 8.B
(normative)

Normative Tables

8.B.1 Huffman tables for SSC

The function ssc_huff_dec() is used as:

data = ssc_huff_dec (t_huff, codeword),

where *t_huff* is the selected Huffman table and *codeword* is the word read from the bitstream. The return value *data*, is a Huffman table index corresponding to a specific code word.

Huffman table overview:

Table 8.B.1 — huff_sgrid

Index	huff_sgrid
0	100001
1	11101
2	11110
3	1100
4	1101
5	1010
6	0111
7	001
8	1011
9	0110
10	1001
11	0101
12	0000
13	0001
14	11100
15	01001
16	111111
17	111110
18	100000
19	010001
20	010000
21	10001

Table 8.B.2 — huff_sampba

index	huff_sampba	index	huff_sampba	index	huff_sampba	index	huff_sampba
0	110010010	64	1101	128	0110	192	110010011001
8	0100111	72	001	136	11000	200	110010011000100
16	1100101	80	000	144	01000	208	110010011000101
24	110011	88	1111	152	010010	216	110010011000110
32	01110	96	1110	160	0100110	224	110010011000111
40	01111	104	1011	168	11001000	232	11001001100000
48	0101	112	1010	176	1100100111	240	11001001100001
56	1001	120	1000	184	11001001101		

Table 8.B.3 — huff_sampbr

index	huff_sampbr	index	huff_sampbr	index	huff_sampbr
-240	111111110110001000010	-72	111101101	96	1111111101100011
-232	111111110110001000011	-64	11110111	104	111111110110101
-224	111111110110001000100	-56	1111010	112	111111110110000
-216	111111110110001000101	-48	111100	120	1111111110110001010001
-208	111111110110001000110	-40	111110	128	1111111110110001010010
-200	111111110110001000111	-32	11101	136	1111111110110001010011
-192	111111110110001001000	-24	0111	144	1111111110110001010100
-184	111111110110001001001	-16	010	152	1111111110110001010101
-176	111111110110001001010	-8	00	160	1111111110110001010110
-168	111111110110001001011	0	10	168	1111111110110001010111
-160	111111110110001001100	8	110	176	1111111110110001011000
-152	111111110110001001101	16	0110	184	1111111110110001011001
-144	111111110110001001110	24	11100	192	1111111110110001011010
-136	111111110110001001111	32	1111110	200	1111111110110001011011
-128	111111110110001010000	40	11111110	208	1111111110110001011100
-120	111111110110100	48	111101100	216	1111111110110001011101
-112	11111111011001	56	1111111110	224	1111111110110001011110
-104	1111111111100	64	11111111010	232	1111111110110001011111
-96	111111110111	72	111111111111	240	1111111011000100000
-88	11111111110	80	1111111111101		
-80	1111111100	88	11111111011011		

Table 8.B.4 — huff_sampca

index	huff_sampca	index	huff_sampca	index	huff_sampca
0	01101101011	88	000	176	0011001
8	01101100	96	1111	184	011011011
16	0011000	104	1110	192	0110110100
24	0110111	112	1100	200	011011010101
32	011010	120	1011	208	0110110101001
40	00111	128	1000	216	01101101010001
48	10011	136	0101	224	01101101010000010
56	0100	144	0010	232	01101101010000011
64	0111	152	10010	240	011011010100000
72	1010	160	01100		
80	1101	168	001101		

Table 8.B.5 — huff_sampcr[0]

index	huff_sampcr[0]	index	huff_sampcr[0]	index	huff_sampcr[0]
-26	111001010111000	-8	0110111	10	01101100
-25	01101101110	-7	001000	11	00011010
-24	1110010101111	-6	111000	12	011011010
-23	000111001011	-5	00101	13	000111010
-22	011010111110	-4	11101	14	1110010111
-21	111001010100	-3	0011	15	0110110110
-20	00011100100	-2	1111	16	0001110011
-19	01101011110	-1	110	17	11100101101
-18	11100101100	0	10	18	01101101111
-17	0001110110	1	010	19	00011101110
-16	0110101110	2	0111	20	111001010110
-15	1110010100	3	0000	21	111001010101
-14	000111000	4	01100	22	011010111111
-13	011010110	5	00010	23	000111001010
-12	00011011	6	001001	24	11100101011101
-11	01101010	7	1110011	25	00011101111
-10	11100100	8	0110100	26	111001010111001
-9	0001111	9	0001100		

Table 8.B.6 — huff_sampcr[1]

index	huff_sampcr[1]	index	huff_sampcr[1]	index	huff_sampcr[1]
-26	100111001011	-8	110110	10	11011111
-24	11011100010	-6	11010	12	10011101
-22	110111101110	-4	1100	14	110111001
-20	11011110110	-2	111	16	1101111010
-18	1101110000	0	0	18	1001110011
-16	100111000	2	101	20	11011100011
-14	110111100	4	1000	22	10011100100
-12	11011101	6	10010	24	110111101111
-10	1001111	8	100110	26	100111001010

Table 8.B.7 — huff_sampcr[2]

index	huff_sampcr[2]	index	huff_sampcr[2]	index	huff_sampcr[2]
-28	01011000101	-8	0100	12	0101110
-24	010110000	-4	00	16	01011010
-20	010110011	0	1	20	010110010
-16	01011011	4	011	24	0101100011
-12	0101111	8	01010	28	01011000100

Table 8.B.8 — huff_sampcr[3]

index	huff_sampcr[3]	index	huff_sampcr[3]	index	huff_sampcr[3]
-32	00010101	-8	01	16	00011
-24	000100	0	1	24	0001011
-16	0000	8	001	32	00010100

Table 8.B.9 — huff_sfreqba

index	huff_sfreqba	index	huff_sfreqba
0	101111110100101100100	1944	110001110
8	101111110100101100101	1952	111111110
16	101111110100101100110	1960	10000101
24	101111110100101100111	1968	00101110
32	101111110100101101000	1976	10000110
40	101111110100101101001	1984	00100011
48	101111110100101101010	1992	11100100
56	101111110100101101011	2000	10011010
64	101111110100101101100	2008	00101111
72	101111110100101101101	2016	111101100
80	101111110100101101110	2024	01111101
88	101111110100101101111	2032	01110100
96	101111110100101110000	2040	111011010
104	101111110100101110001	2048	10111011
112	101111110100101110010	2056	10011011
120	101111110100101110011	2064	10010000
128	101111110100101110100	2072	00110000
136	10111111010011	2080	10111100
144	101111110100101110101	2088	01011011
152	101111110101	2096	111011011
160	10111111011	2104	01001011
168	100010000	2112	10111101
176	001010010	2120	10011100
184	01000010	2128	00110001
192	101111111	2136	110001111
200	111001101	2144	01111110
208	00010001	2152	10010001
216	111001110	2160	10110011
224	111100001	2168	111011100
232	101010101	2176	00001110
240	010000011	2184	10010010
248	1111101001	2192	10111110
256	111100010	2200	10100110
264	111001111	2208	00110010
272	100111011	2216	00001111
280	010111011	2224	00011000
288	1111101010	2232	110101111
296	100010001	2240	110110000
304	0111000011	2248	111111111
312	001010011	2256	11111001
320	1111101011	2264	00111100
328	000100100	2272	00010000
336	000100101	2280	10100111
344	1111101100	2288	01101101
352	000100110	2296	01011100
360	110000000	2304	00000000
368	00110110	2312	00111101
376	010000110	2320	01001100
384	011101010	2328	00011001
392	011101011	2336	00100100
400	00010100	2344	110110001
408	010111100	2352	00111110
416	111100011	2360	00100101
424	110000001	2368	11001000
432	010111101	2376	00100110

440	011101100	2384	00000001
448	011101101	2392	100011101
456	1101001100	2400	110010010
464	1010101001	2408	00110011
472	100111100	2416	10000111
480	00011111	2424	011001011
488	010111110	2432	01111111
496	1111101101	2440	00000010
504	111010000	2448	101101000
512	110000010	2456	01001101
520	00000100	2464	111101101
528	00100000	2472	00100111
536	111010001	2480	101101001
544	111100100	2488	110010011
552	111010010	2496	111101110
560	110000011	2504	111000111
568	01000100	2512	101101010
576	10010101	2520	00111111
584	01010001	2528	100100110
592	10101001	2536	111011101
600	101010110	2544	010011100
608	111100101	2552	110110010
616	011101110	2560	101101011
624	00000101	2568	101001001
632	10001001	2576	01000000
640	01100000	2584	101101100
648	01100001	2592	011110011
656	101010111	2600	100000000
664	100111101	2608	110010100
672	111010011	2616	00011010
680	101011000	2624	01001111
688	10101101	2632	100100111
696	10000010	2640	111001010
704	110100111	2648	100000001
712	00000110	2656	100000010
720	100010100	2664	101101101
728	001010100	2672	110010101
736	110101000	2680	000101111
744	010111111	2688	001101000
752	01010010	2696	111001011
760	011000100	2704	1110011001
768	011101111	2712	1101100110
776	111100110	2720	011011100
784	110101001	2728	000110110
792	111110111	2736	111011110
800	110000100	2744	110010110
808	111010100	2752	111101111
816	011000101	2760	010011101
824	1010110010	2768	0001111011
832	111100111	2776	110010111
840	01110001	2784	010100000
848	11001110	2792	101101110
856	01110010	2800	101010000
864	011110000	2808	1110111110
872	110111001	2816	101010001
880	110101010	2824	100000011
888	101011100	2832	1110111111

896	00101011	2840	010100001
904	01000101	2848	100101000
912	11001111	2856	1111000000
920	01100011	2864	110011000
928	110111100	2872	110110100
936	011001000	2880	1010110011
944	011001001	2888	1101100111
952	011001010	2896	1111110001
960	110111101	2904	001101001
968	101011101	2912	1101101010
976	000100111	2920	11011010110
984	00110111	2928	110011001
992	01100110	2936	00101000
1000	10000011	2944	11011010111
1008	00111000	2952	011011101
1016	011110001	2960	001101010
1024	110000101	2968	1001010010
1032	101011110	2976	1101101100
1040	1111110000	2984	011011110
1048	100111110	2992	00110101100
1056	10111000	3000	000110111
1064	10111001	3008	101101111
1072	11010110	3016	1001010011
1080	11011101	3024	0101110101
1088	111010101	3032	110011010
1096	100111111	3040	1111000001
1104	101000000	3048	0000001100
1112	101011111	3056	1111101000
1120	110000110	3064	1001110100
1128	110111110	3072	01110000101
1136	00000111	3080	0110111110
1144	101000001	3088	110011011
1152	010000111	3096	00110101101
1160	101100000	3104	11011011010
1168	001010101	3112	1101101110
1176	110111111	3120	1011111100
1184	00111001	3128	0000001101
1192	111111001	3136	0110111111
1200	110000111	3144	000111100
1208	101000010	3152	10011101010
1216	111000000	3160	11011011011
1224	101100001	3168	11011011110
1232	101100010	3176	0011010111
1240	111000001	3184	0000001110
1248	111000010	3192	0111000000
1256	01010011	3200	11011011111
1264	00111010	3208	01000001000
1272	111000011	3216	1011111101001011101 10
1280	1101001101	3224	110111000000
1288	111101000	3232	1101110001
1296	110101011	3240	10011101011
1304	111000100	3248	110111000001
1312	01010100	3256	1110011000
1320	111111010	3264	0000001111
1328	11010000	3272	0111000001
1336	01100111	3280	1011111101001011101 11
1344	01101000	3288	110111000010

1352	101000011	3296	010000010010
1360	00001000	3304	01000001010
1368	11000100	3312	010000010011
1376	10001011	3320	10101010000
1384	011110010	3328	0100000101100
1392	111101001	3336	0001111010
1400	10001100	3344	010000010111
1408	111101010	3352	101111110100101111000
1416	01010101	3360	0100000101101
1424	111000101	3368	110111000011
1432	00001001	3376	10101010001
1440	111010110	3384	01011101000
1448	111111011	3392	010111010010
1456	10100010	3400	010111010011
1464	11111000	3408	1011111101000
1472	01101001	3416	101111110100101111001
1480	01010110	3424	011100001000
1488	01101010	3432	101111110100101111010
1496	10001101	3440	101111110100101111011
1504	01101011	3448	101111110100101111100
1512	110101110	3456	101111110100101111101
1520	10010110	3464	101111110100101111110
1528	10010111	3472	101111110100101111111
1536	01010111	3480	011100001001
1544	01000110	3488	101111110100100000000
1552	111000110	3496	101111110100100000001
1560	01101100	3504	101111110100100000010
1568	00001010	3512	101111110100100000011
1576	01111010	3520	101111110100100000100
1584	00101100	3528	101111110100100000101
1592	111101011	3536	101111110100100000110
1600	00111011	3544	101111110100100000111
1608	01110011	3552	101111110100100001000
1616	01111011	3560	101111110100100001001
1624	11010001	3568	101111110100100001010
1632	111111100	3576	101111110100100001011
1640	100010101	3584	101111110100100001100
1648	01011000	3592	101111110100100001101
1656	10011000	3600	101111110100100001110
1664	100011100	3608	101111110100100001111
1672	10100011	3616	101111110100100010000
1680	110001010	3624	101111110100100010001
1688	0001110	3632	101111110100100010010
1696	00010101	3640	101111110100100010011
1704	10110010	3648	101111110100100010100
1712	111010111	3656	101111110100100010101
1720	01011001	3664	101111110100100010110
1728	00100001	3672	101111110100100010111
1736	00001011	3680	101111110100100011000
1744	00100010	3688	101111110100100011001
1752	01000111	3696	101111110100100011010
1760	00001100	3704	101111110100100011011
1768	111011000	3712	101111110100100011100
1776	11000110	3720	101111110100100011101
1784	11010010	3728	101111110100100011110
1792	00010110	3736	101111110100100011111
1800	01001000	3744	101111110100100100000

1808	101001000	3752	10111111010010100001
1816	110001011	3760	10111111010010100010
1824	01001001	3768	10111111010010100011
1832	10111010	3776	10111111010010100100
1840	01111100	3784	10111111010010100101
1848	000101110	3792	10111111010010100110
1856	00001101	3800	10111111010010100111
1864	10000100	3808	10111111010010101000
1872	01011010	3816	10111111010010101001
1880	00101101	3824	10111111010010101010
1888	01001010	3832	10111111010010101011
1896	111011001	3840	10111111010010101100
1904	111111101	3848	10111111010010101101
1912	10001111	3856	10111111010010101110
1920	10011001	3864	10111111010010101111
1928	10100101	3872	10111111010010110000
1936	101100011	3880	10111111010010110001

Table 8.B.10 — huff_sfreqbr

index	huff_sfreqbr	index	huff_sfreqbr
0	0000001011	1944	11000101001010
8	101111	1952	000011011111
16	10101	1960	000011100000
24	11011	1968	010100101101
32	11001	1976	111100101011
40	10100	1984	110001010011
48	10001	1992	010100101110
56	01100	2000	000011100001
64	01101	2008	000011100010
72	01011	2016	010100101111
80	00111	2024	000011100011
88	111101	2032	1100010101000
96	111110	2040	0111000000000
104	111010	2048	0111000000001
112	111001	2056	000011100100
120	100111	2064	010010000100011
128	101100	2072	000011100101
136	011111	2080	011100000001
144	100100	2088	100001111010
152	010101	2096	0111000000100
160	010000	2104	1100010101001
168	001100	2112	000011100110
176	001001	2120	1100010101010
184	000010	2128	000011100111
192	000001	2136	0111000000101
200	000111	2144	000011101000
208	1111110	2152	110001010010110
216	1111000	2160	000011101001
224	1100011	2168	01001000010001011100010
232	1110111	2176	110001010010111
240	1101000	2184	000011101010
248	1001100	2192	11000101010110
256	1011010	2200	11000101010111
264	1100001	2208	0111000000110
272	1011100	2216	0111000000111
280	1000000	2224	1100010101100

288	1000010	2232	0111001101000
296	1001101	2240	1100010101101
304	0111101	2248	11000101011100
312	1001010	2256	0111001101001
320	0100010	2264	0111001101010
328	0111010	2272	000011101011
336	0100011	2280	11000101011101
344	0001011	2288	11000101011110
352	0100101	2296	01001000010001011100011
360	0100110	2304	11000101011111
368	0001100	2312	0111001101011
376	0010100	2320	110001011000000
384	0011010	2328	110001011000001
392	0010000	2336	1100010110001
400	0011011	2344	1100010110010
408	0010101	2352	110001011000010
416	11111110	2360	01001000010001011100100
424	0001000	2368	110001011000011
432	11101101	2376	11000101100110
440	11100000	2384	110001011001110
448	11010101	2392	01001000010001011100101
456	11100010	2400	01001000010001011100110
464	11010110	2408	11000101101000
472	11000001	2416	1000011110110
480	11101100	2424	11000101101001
488	10110110	2432	110001011001111
496	10111011	2440	11000101101010
504	10110111	2448	110001011010110
512	01001001	2456	1100010110110
520	00010011	2464	110001011010111
528	11010010	2472	110001011011100
536	01010011	2480	110001011011101
544	10000110	2488	110001011011110
552	01110110	2496	11000101110000
560	00101110	2504	01001000010001011100111
568	01001110	2512	110001011011111
576	01010000	2520	01001000010001011101000
584	01001111	2528	11000101110001
592	00101101	2536	01001000010001011101001
600	00000011	2544	01001000010001011101010
608	00000001	2552	110001011100100
616	111100111	2560	01001000010001011101011
624	111100110	2568	110001011100101
632	110100110	2576	01001000010001011101100
640	111100100	2584	01001000010001011101101
648	00001111	2592	110001011100110
656	00101111	2600	110001011100111
664	111111110	2608	1100010111010
672	101110101	2616	01001000010001011101110
680	110101110	2624	110001011101100
688	100101100	2632	01001000010001011101111
696	00010100	2640	110001011101101
704	110101111	2648	01001000010001011110000
712	111000011	2656	01001000010001011110001
720	011100011	2664	01001000010001011110010
728	110100111	2672	110001011101110
736	100000100	2680	110001011101111

744	100000101	2688	1100010111110000
752	011101111	2696	1100010111110001
760	011100001	2704	1100010111110010
768	100001110	2712	11000101111010
776	101110100	2720	11000101111011
784	000110110	2728	1100010111110011
792	011100100	2736	1100010111111000
800	000100101	2744	01001000010001011110011
808	001011001	2752	01001000010001011110100
816	1111111110	2760	01001000010001011110101
824	1111001011	2768	11000101111101
832	100101101	2776	01001000010001011110110
840	010010001	2784	1100010111111001
848	001000101	2792	01001000010001011110111
856	010100010	2800	01001000010001011111000
864	1100000001	2808	1100010111111100
872	011110010	2816	01001000010001011111001
880	1001011101	2824	1100010111111101
888	000110111	2832	01001000010001011111010
896	000101010	2840	11000101111111
904	1110001100	2848	01001000010001011111011
912	000000100	2856	01001000010001011111100
920	000011000	2864	1110000111000000
928	1101010011	2872	01001000010001011111101
936	1110000100	2880	1110000111000001
944	000100100	2888	1110000111000010
952	011100010	2896	01001000010001011111110
960	1110001101	2904	01001000010001011111111
968	0111100000	2912	01001000010001000000000
976	000101011	2920	11100011100010
984	0111100001	2928	01001000010001000000001
992	1101010000	2936	1110000111000011
1000	0111000001	2944	1110000111000110
1008	1101010001	2952	01001000010001000000010
1016	0111001010	2960	1110000111000111
1024	0111001011	2968	00000010100000
1032	1000011111	2976	01001000010001000000011
1040	1100000010	2984	01001000010001000000100
1048	0010001001	2992	01001000010001000000101
1056	1101010010	3000	01001000010001000000110
1064	1100000000	3008	01001000010001000000111
1072	0111001110	3016	01001000010001000001000
1080	1110001111	3024	00000010100001
1088	111111111110	3032	01001000010001000001001
1096	0000111011	3040	01001000010001000001010
1104	0111100010	3048	01001000010001000001011
1112	0111100011	3056	01001000010001000001100
1120	0001101000	3064	01001000010001000001101
1128	0111001100	3072	00000010100010
1136	0111100111	3080	01001000010001000001110
1144	1001011110	3088	01001000010001000001111
1152	0010001100	3096	01001000010001000010000
1160	110000001100	3104	00000010100011
1168	0111001111	3112	01001000010001000010001
1176	111111111111	3120	01001000010000
1184	0111011100	3128	01001000010001000010010
1192	0010001101	3136	01001000010001000010011

1200	1000001100	3144	0100100001000100010100
1208	0010001110	3152	0100100001000100010101
1216	0010110001	3160	0100100001000100010110
1224	0111011101	3168	0100100001000100010111
1232	10010111110	3176	0100100001000100011000
1240	0100100000	3184	0100100001000100011001
1248	01111001101	3192	0100100001000100011010
1256	0000000000	3200	0100100001000100011011
1264	1001011100	3208	0100100001000100011100
1272	0001101001	3216	0100100001000100011101
1280	0000000001	3224	0100100001000100011110
1288	0000000010	3232	0100100001000100011111
1296	01001000011	3240	0100100001000100100000
1304	11100001010	3248	0100100001000100100001
1312	110000001101	3256	0100100001000100100010
1320	0001101010	3264	0100100001000100100011
1328	0000000011	3272	0100100001000100100100
1336	0001101011	3280	0100100001000100100101
1344	01110011011	3288	0100100001000100100110
1352	10010111111	3296	0100100001000100100111
1360	01010001100	3304	0100100001000100101000
1368	11100011101	3312	0100100001000100101001
1376	01010001101	3320	0100100001000100101010
1384	110000001110	3328	0100100001000100101011
1392	1000011110111	3336	0100100001000100101100
1400	11100001011	3344	0100100001000100101101
1408	11000100000	3352	0100100001000100101110
1416	01111001100	3360	0100100001000100101111
1424	00100011110	3368	0100100001000100110000
1432	0010001000	3376	0100100001000100110001
1440	11000100001	3384	0100100001000100110010
1448	11000100010	3392	0100100001000100110011
1456	01010001110	3400	0100100001000100110100
1464	01010001111	3408	0100100001000100110101
1472	0000110010	3416	0100100001000100110110
1480	111000111001	3424	0100100001000100110111
1488	00100011111	3432	0100100001000100111000
1496	10000011010	3440	0100100001000100111001
1504	11000100011	3448	0100100001000100111010
1512	00000010101	3456	0100100001000100111011
1520	10000011011	3464	0100100001000100111100
1528	111100101000	3472	0100100001000100111101
1536	10000011100	3480	0100100001000100111110
1544	01010010000	3488	0100100001000100111111
1552	010010000101	3496	0100100001000101000000
1560	01010010001	3504	0100100001000101000001
1568	110000001111	3512	0100100001000101000010
1576	00001100110	3520	0100100001000101000011
1584	0100100001001	3528	0100100001000101000100
1592	0101001001000	3536	0100100001000101000101
1600	110001001000	3544	0100100001000101000110
1608	00001100111	3552	0100100001000101000111
1616	100000111010	3560	0100100001000101001000
1624	1100010010010	3568	0100100001000101001001
1632	110001001010	3576	0100100001000101001010
1640	00101100000	3584	0100100001000101001011
1648	010100100101	3592	0100100001000101001100

1656	00001101000	3600	0100100001000101001101
1664	100000111011	3608	0100100001000101001110
1672	01010010011	3616	0100100001000101001111
1680	10000011110	3624	0100100001000101010000
1688	00101100001	3632	0100100001000101010001
1696	00001101001	3640	0100100001000101010010
1704	110001001011	3648	0100100001000101010011
1712	11000100110	3656	0100100001000101010100
1720	110001001110	3664	0100100001000101010101
1728	110001001111	3672	0100100001000101010110
1736	111100101001	3680	0100100001000101010111
1744	100000111110	3688	0100100001000101011000
1752	00001101010	3696	0100100001000101011001
1760	1100010010011	3704	0100100001000101011010
1768	00001101011	3712	0100100001000101011011
1776	00001101100	3720	0100100001000101011100
1784	00001101101	3728	0100100001000101011101
1792	110001010000	3736	0100100001000101011110
1800	100000111111	3744	0100100001000101011111
1808	000000101001	3752	0100100001000101100000
1816	1100010100010	3760	0100100001000101100001
1824	100001111000	3768	0100100001000101100010
1832	0101001001001	3776	0100100001000101100011
1840	010100101000	3784	0100100001000101100100
1848	111100101010	3792	0100100001000101100101
1856	000011011100	3800	0100100001000101100110
1864	010100101001	3808	0100100001000101100111
1872	0101001010100	3816	0100100001000101101000
1880	100001111001	3824	0100100001000101101001
1888	1100010100011	3832	0100100001000101101010
1896	010100101011	3840	0100100001000101101011
1904	000011011101	3848	0100100001000101101100
1912	1100010100100	3856	0100100001000101101101
1920	010100101100	3864	0100100001000101101110
1928	0101001010101	3872	0100100001000101101111
1936	000011011110	3880	0100100001000101110000

Table 8.B.11 — huff_sfreqc

index	huff_sfreqc	index	huff_sfreqc
0	00100011011100011010010000	1944	011101111
8	00100011011100011010010001	1952	11101110
16	00100011011100011010010010	1960	10011100
24	00100011011100011010010011	1968	11010011
32	00100011011100011010010100	1976	00011011
40	00100011011100011010010101	1984	110010111
48	00100011011100011010010110	1992	111110001
56	00100011011100011010010111	2000	11001110
64	00100011011100011010011000	2008	10011001
72	00100011011100011010011001	2016	111111100
80	0010001101110011011	2024	00001100
88	00100011011100011010011010	2032	111010100
96	0010001101110011100	2040	01000101
104	0010001101110010	2048	11110000
112	0010001101110011 1	2056	0100111
120	100110101011010	2064	00001110
128	100110101011000	2072	01000010

136	100110101011001	2080	111111101
144	0010001101100	2088	111101100
152	0010001101111	2096	00011100
160	011001001001	2104	01110101
168	111101000111	2112	01100000
176	11000000000	2120	10001111
184	0111000010	2128	10111110
192	0010011010	2136	10010101
200	11100010010	2144	1010010
208	0000101111	2152	11100100
216	0000001100	2160	01010111
224	10110110111	2168	00101010
232	11101011000	2176	01110011
240	11011100010	2184	10101011
248	10111101000	2192	11011000
256	10101001000	2200	01111101
264	01101011101	2208	00101011
272	10001011101	2216	00110101
280	11001101101	2224	11000011
288	0101100011	2232	10001001
296	001011100	2240	111100110
304	0000001000	2248	111100010
312	11000000001	2256	10010100
320	01100100101	2264	11010111
328	10011010001	2272	11011101
336	0001100010	2280	11111111
344	001010000	2288	10111000
352	110001001	2296	10110111
360	01101011100	2304	11000111
368	10101110100	2312	01100110
376	0000000001	2320	11010101
384	1000001101	2328	10110011
392	0111000011	2336	111101010
400	0011101010	2344	110111100
408	000110010	2352	01001010
416	1111011110	2360	10000000
424	11101101001	2368	1010110
432	01101000000	2376	0001001
440	10001010001	2384	01010101
448	111101000110	2392	01010011
456	00100011010	2400	01111001
464	11011100000	2408	10010110
472	0000001101	2416	10101000
480	1001001011	2424	01011101
488	11100111000	2432	11010001
496	01101000001	2440	11001100
504	11101011001	2448	00100000
512	1011001011	2456	10111010
520	1101010011	2464	0001111
528	1100010000	2472	11101000
536	0000100010	2480	00101111
544	0000101110	2488	01110110
552	0110101101	2496	11100101
560	1001110111	2504	10101010
568	001110100	2512	10011111
576	011100011	2520	11111011
584	001000111	2528	10001101

592	110000011	2536	10000111
600	111111001	2544	11101100
608	10110110110	2552	0100000
616	11110100010	2560	01000011
624	1000001100	2568	111101011
632	011010010	2576	111101001
640	100001010	2584	00111101
648	100100110	2592	10101111
656	1010001011	2600	0010110
664	0000100011	2608	11101001
672	11001111111	2616	10011000
680	1100000001	2624	11001001
688	01011100	2632	10110100
696	00000111	2640	01010001
704	110001101	2648	10100110
712	0100100111	2656	10100011
720	1010100101	2664	00110110
728	001100011	2672	10001100
736	0100100110	2680	01111111
744	0101000010	2688	01001011
752	010100000	2696	0111101
760	000110000	2704	11010000
768	011111100	2712	101111011
776	110000001	2720	110110101
784	00110111	2728	01011010
792	00000100	2736	10001000
800	11001111110	2744	11000010
808	10001010000	2752	11100001
816	11011100001	2760	10100000
824	11100010011	2768	01011001
832	111100011	2776	111110011
840	100001011	2784	10010111
848	111000110	2792	00111000
856	1011101111	2800	01011110
864	11010100100	2808	000010110
872	0110101100	2816	110010100
880	0110010011	2824	11100110
888	1001001010	2832	01100111
896	1111011111	2840	111010111
904	100100000	2848	00001101
912	00111100	2856	111110100
920	1100111110	2864	110101000
928	1000101111	2872	110101101
936	1011001010	2880	11100000
944	00010100	2888	111101110
952	1111011010	2896	00010101
960	10100010100	2904	00101001
968	10101110101	2912	011011111
976	11001101000	2920	011100000
984	0001100011	2928	10000001
992	000110011	2936	10111111
1000	010110000	2944	000000101
1008	100000111	2952	101111000
1016	010101000	2960	01100011
1024	0110101111	2968	111110010
1032	10111101001	2976	111010101
1040	11101101000	2984	10100001

1048	11001101100	2992	101101100
1056	0111000100	3000	01000100
1064	011101110	3008	110010000
1072	00110011	3016	100010110
1080	01111000	3024	00100001
1088	1001110110	3032	0011111
1096	11011100011	3040	01011011
1104	0101000011	3048	011111101
1112	1011110101	3056	01101110
1120	1111110000	3064	01010110
1128	1101100111	3072	10111001
1136	1110001000	3080	111000111
1144	011100100	3088	10110000
1152	01001101	3096	110011110
1160	011001000	3104	111110101
1168	0111001010	3112	111110000
1176	1000101001	3120	00100111
1184	110110010	3128	01001000
1192	10110001	3136	011000100
1200	0111001011	3144	101111001
1208	10011010000	3152	00100010
1216	00111010111	3160	01110100
1224	10100010101	3168	110010101
1232	11100111001	3176	101001111
1240	1100100011	3184	00100100
1248	101001110	3192	01100101
1256	100111010	3200	101101011
1264	111100111	3208	101010011
1272	001011101	3216	110101100
1280	1100010001	3224	001100010
1288	1111011011	3232	000010100
1296	1010111001	3240	1110101101
1304	0011001010	3248	1001101001
1312	0111000101	3256	1101100110
1320	001001100	3264	0010001100
1328	011011110	3272	1100100010
1336	11010010	3280	0000001001
1344	00001111	3288	000000111
1352	1010111000	3296	10101001001
1360	000110100	3304	10011010100
1368	010101001	3312	0110100001
1376	110111111	3320	00000110
1384	01000110	3328	0000101011
1392	1001101011	3336	1011101110
1400	1110011101	3344	0000000000
1408	010010010	3352	0011001011
1416	011010001	3360	11010100101
1424	101000100	3368	10001011100
1432	001010001	3376	1100110111
1440	011000101	3384	1011011010
1448	101110110	3392	11001101001
1456	10011110	3400	00001010101
1464	10010001	3408	00001010100
1472	000010000	3416	0010001101101
1480	1010111011	3424	0101100010
1488	1100110101	3432	0010011011
1496	1111110001	3440	011001001000

1504	010111111	3448	001000110111000
1512	1111010000	3456	10011010101110
1520	111100101	3464	10011010101111
1528	111100100	3472	1001101010101
1536	0001011	3480	00111010110
1544	01001100	3488	001000110111010
1552	000110101	3496	1001101010100
1560	1110110101	3504	0010001101110011010011011
1568	010111110	3512	0010001101110011010011100
1576	101100100	3520	1001101010110111
1584	110010110	3528	001000110111011
1592	100100111	3536	0010001101110011010011101
1600	111011011	3544	1001101010110110
1608	00001001	3552	0010001101110011010011110
1616	01101010	3560	0010001101110011010011111
1624	11111101	3568	0010001101110011010100000
1632	111001111	3576	0010001101110011010100001
1640	000000001	3584	0010001101110011010100010
1648	00111001	3592	0010001101110011010100011
1656	00100101	3600	0010001101110011010100100
1664	11000101	3608	0010001101110011010100101
1672	10001110	3616	0010001101110011010100110
1680	001100100	3624	0010001101110011010100111
1688	110110100	3632	0010001101110011010101000
1696	01100001	3640	0010001101110011010101001
1704	111011110	3648	0010001101110011010101010
1712	100100100	3656	0010001101110011010101011
1720	101101010	3664	0010001101110011010101100
1728	111000101	3672	0010001101110011010101101
1736	00011101	3680	0010001101110011010101110
1744	01111100	3688	0010001101110011010101111
1752	10000100	3696	0010001101110011010110000
1760	00000001	3704	0010001101110011010110001
1768	00000101	3712	0010001101110011010110010
1776	111011111	3720	0010001101110011010110011
1784	00111011	3728	0010001101110011010110100
1792	110000010	3736	0010001101110011010110101
1800	110001100	3744	0010001101110011010110110
1808	100010101	3752	0010001101110011010110111
1816	00010000	3760	0010001101110011010111000
1824	00010001	3768	0010001101110011010111001
1832	0110110	3776	0010001101110011010111010
1840	10000110	3784	0010001101110011010111011
1848	100100001	3792	0010001101110011010111100
1856	011010011	3800	0010001101110011010111101
1864	01000111	3808	0010001101110011010111110
1872	00110000	3816	0010001101110011010111111
1880	10000010	3824	0010001101110011101000000
1888	00110100	3832	0010001101110011101000001
1896	110111101	3840	0010001101110011101000010
1904	110111001	3848	0010001101110011101000011
1912	01010010	3856	0010001101110011101000100
1920	10011011	3864	0010001101110011101000101
1928	11011011	3872	0010001101110011101000110
1936	110111110	3880	0010001101110011101000111

Table 8.B.12 — huff_nlag

index	huff_nlag	index	huff_nlag
-512	1000101011011011100	4	110
-508	1000101011011011101	8	1111
-504	1000101011011011110	12	10010
-500	1000101011011011111	16	100001
-496	1000101110100000000	20	1000100
-492	1000101110100000001	24	10000000
-488	1000101110100000010	28	100000010
-484	1000101110100000011	32	1000101000
-480	1000101110100000100	36	10001010011
-476	1000101110100000101	40	100010111001
-472	1000101110100000110	44	100010101110
-468	1000101110100000111	48	1000101011010
-464	1000101110100001000	52	1000101011110
-460	1000101110100001001	56	10001010110111
-456	1000101110100001010	60	1000101001000
-452	1000101110100001011	64	10001010110011
-448	1000101110100001100	68	100010101111100
-444	1000101110100001101	72	10001010010010
-440	1000101110100001110	76	100010111011110
-436	1000101110100001111	80	100010100101110
-432	1000101110100010000	84	10001011101110110
-428	1000101110100010001	88	100010101111101
-424	1000101110100010010	92	100010100101111
-420	1000101110100010011	96	1000101110111110
-416	1000101110100010100	100	10001011101110111
-412	1000101110100010101	104	1000101110111111
-408	1000101110100010110	108	1000101011011000
-404	1000101110100010111	112	1000101110111001111
-400	1000101110100011000	116	1000101011011001
-396	1000101110100011001	120	10000001100000000
-392	1000101110100011010	124	1000101011011010
-388	1000101110100011011	128	10000001100000001
-384	1000101110100011100	132	10000001100000010
-380	1000101110100011101	136	10000001100000011
-376	1000101110100011110	140	10000001100000100
-372	1000101110100011111	144	10000001100000101
-368	1000101110100100000	148	10000001100000110
-364	1000101110100100001	152	10000001100000111
-360	1000101110100100010	156	10000001100001000
-356	1000101110100100011	160	10000001100001001
-352	1000101110100100100	164	1000000110000101
-348	1000101110100100101	168	10000001100001100
-344	1000101110100100110	172	10000001100001101
-340	1000101110100100111	176	1000000110000111
-336	1000101110100101000	180	10000001100010000
-332	1000101110100101001	184	10000001100010001
-328	1000101110100101010	188	10000001100010010
-324	1000101110100101011	192	1000000110001010
-320	1000101110100101100	196	10000001100010011
-316	1000101110100101101	200	10000001100010110
-312	1000101110100101110	204	10000001100010111
-308	1000101110100101111	208	10000001100011000
-304	1000101110100110000	212	10000001100011001
-300	1000101110100110001	216	10000001100011010
-296	1000101110100110010	220	1000000110001110

-292	1000101111010110011	224	10000001100011011
-288	1000101111010110100	228	10000001100011110
-284	1000101111010110101	232	10000001100011111
-280	1000101111010110110	236	10000001100100000
-276	1000101111010110111	240	10000001100100001
-272	1000101111010111000	244	10000001100100010
-268	1000101111010111001	248	10000001100100011
-264	1000101111010111010	252	10000001100100100
-260	1000101111010111011	256	10000001100100101
-256	1000101111010111100	260	10000001100100110
-252	1000101111010111101	264	10000001100100111
-248	1000101111010111110	268	10000001100101000
-244	1000101111010111111	272	10000001100101001
-240	1000101111011000000	276	10000001100101010
-236	1000101111011000001	280	10000001100101011
-232	1000101111011000010	284	10000001100101100
-228	1000101111011000011	288	10000001100101101
-224	1000101111011000100	292	10000001100101110
-220	1000101111011000101	296	10000001100101111
-216	1000101111011000110	300	10000001100110000
-212	1000101111011000111	304	10000001100110001
-208	1000101111011001000	308	10000001100110010
-204	1000101111011001001	312	10000001100110011
-200	10001011101100101	316	10000001100110100
-196	1000101111011001100	320	10000001100110101
-192	1000101111011001101	324	10000001100110110
-188	1000101111011001110	328	10000001100110111
-184	1000101111011001111	332	10000001100111000
-180	1000101111011010000	336	10000001100111001
-176	1000101111011010001	340	10000001100111010
-172	1000101111011010010	344	10000001100111011
-168	1000101111011010011	348	10000001100111100
-164	1000101111011010100	352	10000001100111101
-160	10001011101101011	356	10000001100111110
-156	1000101001011001	360	10000001100111111
-152	1000101111011010101	364	10000001101000000
-148	1000101111011011000	368	10000001101000001
-144	1000101111011011001	372	10000001101000010
-140	1000101111011011010	376	10000001101000011
-136	1000101111011011011	380	10000001101000100
-132	10001011101101110	384	10000001101000101
-128	10001011101101111	388	10000001101000110
-124	1000101111011100000	392	10000001101000111
-120	1000101111011100001	396	10000001101001000
-116	1000101111011100010	400	10000001101001001
-112	1000101111011100011	404	10000001101001010
-108	1000101111011100100	408	10000001101001011
-104	1000101111011100101	412	10000001101001100
-100	1000101111011100110	416	10000001101001101
-96	1000101001011010	420	10000001101001110
-92	100010100100111	424	10000001101001111
-88	1000101001011011	428	10000001101010000
-84	1000101110111010	432	10000001101010001
-80	10001010010100	436	10000001101010010
-76	10001010010101	440	10000001101010011
-72	10001010110010	444	10000001101010100
-68	10000001101100	448	10000001101010101

-64	10000001101101	452	10000001101010110
-60	10001010111111	456	10000001101010111
-56	1000000110111	460	10000001101011000
-52	1000101011000	464	10000001101011001
-48	100010101010	468	10000001101011010
-44	100010101011	472	10000001101011011
-40	100010111000	476	10000001101011100
-36	10001010100	480	10000001101011101
-32	1000000111	484	10000001101011110
-28	1000101111	488	10000001101011111
-24	100010110	492	10001010010011000
-20	1000001	496	10001010010011001
-16	100011	500	10001010010011010
-12	10011	504	10001010010011011
-8	1110	508	10001010010110000
-4	101	512	10001010010110001
0	0		

Table 8.B.13 — huff_nlsf

index	huff_nlsf	index	huff_nlsf	index	huff_nlsf
7	10110011	14	1000	21	1011010
8	101101110	15	00	22	10010011
9	101101111	16	11	23	100100101
10	10110010	17	01	24	100100100
11	1001000	18	1010	25	10110110
12	1011000	19	10111		
13	100101	20	10011		

Note: Values smaller than value 8 are represented by 'escape' value 7 followed by a 3 bits unsigned integer containing the real value. Values greater than value 24 are represented by 'escape' value 25 followed by an 8 bits unsigned integer containing the real value.

Table 8.B.14 — huff_ngain

index	huff_ngain	index	huff_ngain	index	huff_ngain
-13	010011100	-4	0100010	5	01000111
-12	010011011001	-3	010010	6	010011001
-11	010011101110	-2	01011	7	0100110111
-10	010011101011	-1	00	8	0100110100
-9	01001110100	0	1	9	01001101101
-8	01001110110	1	011	10	010011101111
-7	0100110101	2	01010	11	010011101010
-6	010011000	3	010000	12	010011011000
-5	01000110	4	01001111		

Note: Values outside the range [-12 ..12] are represented by 'escape' value –13 followed by an 8 bits signed integer containing the real value.

Table 8.B.15 — huff_scont

index	huff_scont	index	huff_scont	index	huff_scont
0	1000	4	001	8	0000
1	0100	5	1001	9	11
2	101	6	0101		
3	011	7	0001		

Table 8.B.16 — huff_nrofbirths

Index	huff_nrofbirths	Index	huff_nrofbirths	Index	huff_nrofbirths
0	010	21	000001001	42	000000100001110
1	1010	22	00000101	43	000000100001111
2	1110	23	000001100	44	000000100010000
3	011	24	00000001	45	000000100010001
4	110	25	0000010001	46	000000100010010
5	100	26	0000011010	47	000000100010011
6	001	27	0000011011	48	000000100010100
7	1111	28	000000100000100	49	000000100010101
8	0001	29	0000011100	50	000000100010110
9	10111	30	000000100000101	51	000000100010111
10	10110	31	000000100000110	52	000000100011000
11	0000111	32	000000100000111	53	000000100011001
12	000010	33	0000011101	54	000000100011010
13	0000110	34	000000100001000	55	000000100011011
14	00000011	35	0000011110	56	000000100011100
15	00000000	36	000000100001001	57	000000100011101
16	000000101	37	000000100001010	58	000000100011110
17	0000001001	38	0000011111	59	000000100011111
18	0000010000	39	000000100001011	60	00000010000000
19	000000100000010	40	000000100001100		
20	000000100000011	41	000000100001101		

Table 8.B.17 — huff_iid_df[0] and huff_iid_dt[0]

Index	huff_iid_df[0]	huff_iid_dt[0]	Index	huff_iid_df[0]	huff_iid_dt[0]
-30	011111111010110100	0100111011010100	1	010	00
-29	011111111010110101	0100111011010101	2	0001	01011
-28	011111110101110110	0100111011001110	3	01101	010010
-27	011111110101110111	0100111011001111	4	011101	0100001
-26	011111110101110100	0100111011001100	5	0111101	01001100
-25	011111110101110101	0100111011010110	6	01111101	010011011
-24	011111111010001010	0100111011011000	7	011111100	0100111010
-23	011111111010001011	0100111101000110	8	0111111100	01001111001
-22	011111111010001000	0100111101100000	9	01111111100	01001110000
-21	011111110100000000	010011100011000	10	01111110100	010011101111
-20	011111111010110110	010011100011001	11	011111101011	010011100010
-19	011111110100000010	010011101100100	12	0111111101010	0100111101010
-18	011111110101111000	010011101100101	13	01111111101010	0100111011000
-17	011111110100000010	010011101101101	14	01111111010110	01001111010111
-16	011111111010101110	010011110110001	15	011111111010000	01001111010000
-15	011111110101111	01001110110111	16	0111111110101111	010011110110010
-14	01111111010001	01001111010110	17	0111111101000011	010011110100010
-13	01111111101001	0100111000111	18	01111111010111001	010011100011010
-12	0111111101001	0100111101001	19	01111111010000011	010011100011011
-11	011111101010	0100111101101	20	011111111010110111	0100111101100110
-10	011111111011	010011101110	21	01111111010000001	0100111101100111
-9	01111111011	010011110111	22	011111111010001001	0100111101100001
-8	0111111011	01001111000	23	011111111010001110	0100111101000111
-7	0111111111	0100111001	24	011111111010001111	0100111011011001
-6	01111100	010011010	25	011111111010001100	0100111011010111
-5	0111100	010011111	26	011111111010001101	0100111011001101
-4	011100	0100000	27	011111111010110010	0100111011010010
-3	01100	010001	28	011111111010110011	0100111011010011
-2	0000	01010	29	011111111010110000	0100111011010000

-1	001	011	30	011111111010110001	0100111011010001
0	1	1			

Table 8.B.18 — huff_iid_df[1] and huff_iid_dt[1]

Index	huff_iid_df[1]	huff_iid_dt[1]
-14	11111111111111011	1111111111111111001
-13	11111111111111100	1111111111111111010
-12	11111111111111101	1111111111111111011
-11	11111111111111010	11111111111111111000
-10	1111111111111100	11111111111111111001
-9	111111111111100	11111111111111111010
-8	1111111111101	11111111111111101
-7	1111111110	111111111111110
-6	111111110	111111111110
-5	1111110	1111111110
-4	111100	11111110
-3	11101	111110
-2	1101	1110
-1	101	10
0	0	0
1	100	110
2	1100	11110
3	11100	1111110
4	111101	111111110
5	111110	11111111110
6	11111110	1111111111110
7	11111111110	11111111111110
8	1111111111100	11111111111111100
9	11111111111100	111111111111111000
10	11111111111101	111111111111111011
11	111111111111101	111111111111111100
12	11111111111111110	111111111111111101
13	111111111111111110	111111111111111110
14	111111111111111111	111111111111111111

Table 8.B.19 — huff_icc_dt and huff_icc_df

Index	huff_icc_df	huff_icc_dt
-7	11111111111111	11111111111110
-6	11111111111110	1111111111110
-5	111111111110	11111111110
-4	1111111110	111111110
-3	1111110	1111110
-2	11110	11110
-1	110	110
0	0	0
1	10	10
2	1110	1110
3	111110	111110
4	11111110	11111110
5	111111110	1111111110
6	11111111110	111111111110
7	1111111111110	11111111111111

Table 8.B.20 — huff_ipd_df and huff_ipd_dt

Index	huff_ipd_df	huff_ipd_dt
0	1	1
1	000	010
2	0110	0010
3	0100	00011
4	0010	00010
5	0011	0000
6	0101	0011
7	0111	011

Table 8.B.21 — huff_opd_df and huff_opd_dt

Index	huff_opd_df	huff_opd_dt
0	1	1
1	001	010
2	0110	0001
3	0100	00111
4	01111	00110
5	01110	0000
6	0101	0010
7	000	011

Annex 8.C
(informative)

Encoder description

This annex describes the encoder part of the parametric audio-coding algorithm.

8.C.1 Technical overview

The parametric stereo coding algorithm consists of a parametric core that encodes a monaural signal M and a stereo tool that encodes the stereo image information (see Figure 8.C.1). In the case of (dual) mono coding, the parametric stereo tool is inactive.

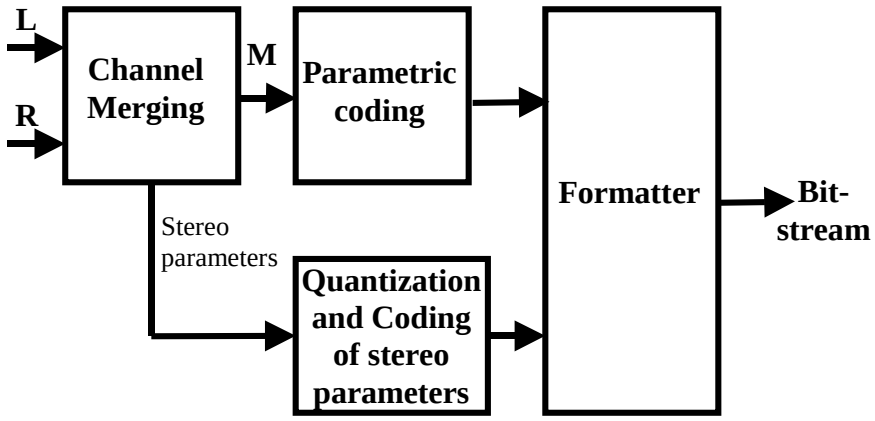


Figure 8.C.1 — Graphical overview of the parametric stereo encoding algorithm

The basic idea behind the monaural parametric audio-coding algorithm is the notion that any audio signal can be described by (or decomposed into) a number of relevant auditory events. These events can then be coded in a parsimonious manner where the parameters are directly linked to psycho-acoustic masking rules and allow manipulations in an auditory meaningful way. Three of these relevant auditory events are identified: transients, sinusoids and noise.

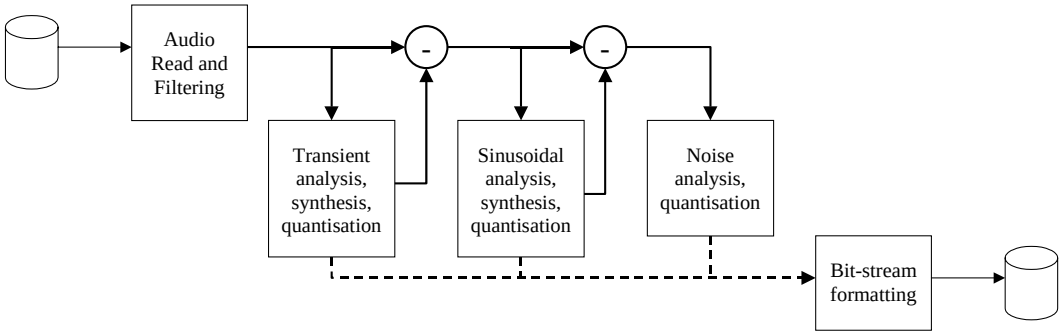


Figure 8.C.2 — functional block diagram of the encoder

The monaural audio encoding scheme can be represented as shown in Figure 8.C.2. The encoding process basically consists of three blocks, responsible for detection and evaluation of above mentioned auditory events.

Each of these blocks again consists of 3 basic parts. First, the **analysis** module estimates specific parameters of the input signal; the **synthesis** module reconstructs a synthesized signal and subtracts it from its input; finally the **quantization** module quantizes parameters, preparing them for the output data stream. In the following sections, it will be shown how the encoder deals with the extraction and coding of each of these components.

1) *Audio read and filtering*: this block reads audio data from an incoming data stream and filters it with a high pass filter to remove inaudible low frequency tones and DC.

2) *Transient processing block*: this block extracts transient (suddenly changing) information from the input audio data, leaving a quasi-stationary signal for further analysis. The signal is analyzed to detect transient components and estimates their corresponding parameters. With these parameters, a 'transient only' audio signal is reconstructed and subtracted from the original input signal.

3) *Sinusoidal processing block*: this block extracts sinusoidal information from the input audio data, leaving a noisy signal for further analysis. The signal is analyzed to detect sinusoidal components and to estimate their parameters. From the sinusoidal parameters, a 'sinusoid only'-signal is reconstructed and subtracted from the residual audio signal after transient analysis.

4) *Noise processing block*: this block estimates the spectral and temporal envelope parameters to match the noise spectrum of the residual audio signal after sinusoid analysis.

5) *Bit-stream formatter*: this block codes the parameters of transients, sinusoids and noise into a bit-stream.

The encoder is organized as a cascade of processing blocks or modules. The modules work sequentially. Each module performs specific operations on an input data buffer and fills an output data buffer. The encoder gets a new piece of data from the incoming data stream and puts it as input for the first processing module. Then it takes an output of this module and feeds it to the next one, and so forth. Output of the last module in the sequence contains all processed data that should be placed in the output data stream. Then this sequence should be repeated, until there is no more new data in the incoming data stream.

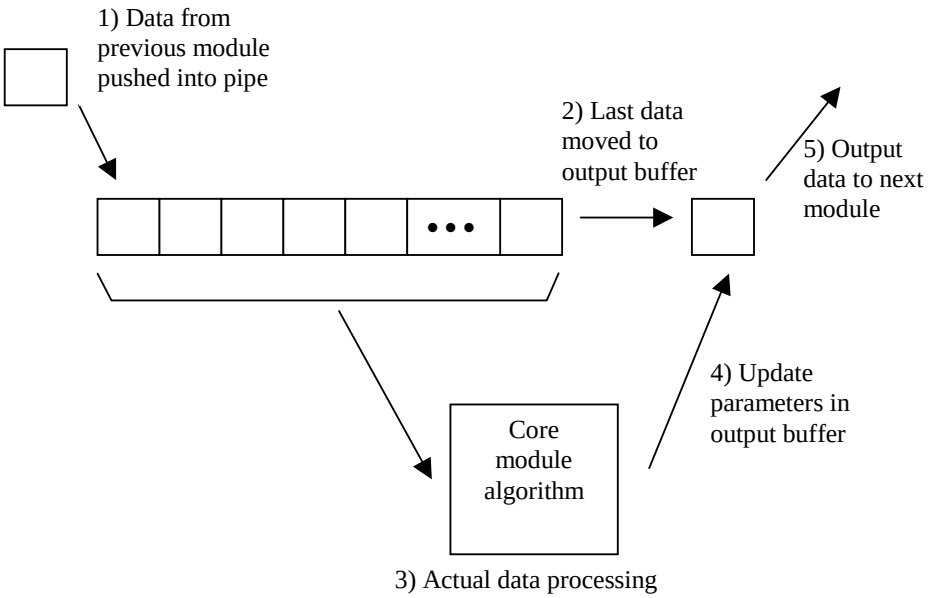


Figure 8.C.3 — Execution sequence inside a module

As it was already mentioned, the encoder does not process the whole input audio stream at once. Instead it breaks the audio stream into sub-frames of *S* samples (approximately 8ms at 44.1kHz) and processes them one by one. If a module requires more data, it buffers the sub-frames until it has enough to proceed with the calculations; a schematic overview of the execution sequence inside a module is show in Figure 8.C.3. A module releases the most recently finished sub-frame for further processing.

8.C.2 Audio read and filtering

Very low frequencies (i.e. < 5 Hz) are inaudible for the human ear. These components are not taken into account for further analysis. This module uses a first order, high pass Butterworth filter to filter off such very low frequencies. Its transfer function is

$$H(z) = \frac{-0.999643937167571 + 0.999643937167571 \cdot z^{-1}}{-0.999287874335142 + z^{-1}}.$$

8.C.3 Transients

The transient module consists of several processes, working sequentially.

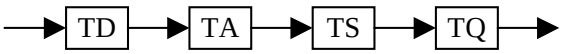


Figure 8.C.4 — Transient processing blocks

8.C.3.1 Transient detection (TD)

The transient detection module analyses the input audio-signal and tries to detect transient positions in the input signal. The algorithm used is the same as in [HILN].

First, it determines an envelope from the input signal. Let *x*[*n*] hold the audio sample at time *n*, λ hold the envelope decay and *envelope*[*n*] hold the envelope curve at time *n.* The envelope is calculated as follows:

$$envelope[n] = \max(\lambda \cdot envelope[n-1], |x[n]|).$$

The envelope decay for transient detection is set to λ = 0.998. If the envelope shows a fivefold increase between adjacent samples, the module marks the first of those two samples as a transient position. After the transient position, there is a hold off period of *S* samples because only one transient per sub-frame is allowed.

For every transient found, the module records the location and sets a flag in the corresponding sub-frame. The location of a transient is relative to the start of the sub-frame.

8.C.3.2 Transient analysis (TA)

Currently two types of transient are supported:

- Meixner transient: the Meixner transient aims to indicate and model certain changes in the audio signal envelope. This transient type can model typical attacks. This transient type is characterized by the following parameters:
  - its position,
  - the attack and decay of the transient envelope (see section 8.6.1.2),

- underlying sinusoids (see section 8.6.1.3),

- Step transient: the step transient aims to indicate and model sudden changes in the audio signal amplitude. The step transient type is only characterized by its position.

The transient analysis module analyses two sub-frames of audio data. First the module calculates an envelope the same way as for transient detection, but the envelope decay for transient analysis is set to λ = 0.98. It then tries to fit all 16 possible Meixner envelopes to this envelope. See Table 8.22 for the possible combinations of *b* and ξ. The sum of the squared error between the Meixner envelope and the audio envelope gives a match error estimate. The combination of *b* and ξ that gives the least error and that does not exceed an error threshold will constitute a Meixner transient, otherwise it will be degraded to a step transient.

In addition for the 'Meixner' type, the module determines up to six sinusoidal parameters under the Meixner envelope. The procedure for extracting the partials is the same as the one used in the sinusoid analysis module. The window used for the regression part of the algorithm is the Meixner envelope. For further information on the extraction of sinusoids, see section 8.C.4.1.

8.C.3.3 Transient synthesis (TS)

The transient synthesis module reconstructs the waveform of the Meixner transient. It synthesizes a pure transient signal from previously obtained Meixner parameters and subtracts it from the incoming signal in order to create a residual signal without transient components for subsequent use by sinusoidal and noise modules. In case of a step transient, this module does not change the residual signal from the previous module.

8.C.3.4 Transient quantisation (TQ)

The floating-point parameters must be quantized and coded before transmission. This module quantises transient parameters – sinusoidal parameters of a Meixner envelope *f*, *a*, φ. The *b* and ξ parameters are already quantized according to Table 8.22. The output of this module is a set of transient parameters that are suitable for the bit-stream format.

8.C.3.4.1 Meixner frequency parameters

This parameter is only present for Meixner type transients. Let *tf* hold the frequency of the nth sinusoid under the Meixner transient in sub-frame sf of channel ch. The frequency is expressed in radians in the range [0, π>.

To convert to representation levels, the encoder quantises the floating-point frequencies on an ERB-scale, with 11.4 representation levels per ERB. The representation level $tf_{rl}[sf][ch][n]$ is calculated as follows

$$tf_{rl}[sf][ch][n] = \lfloor 11.4 \cdot tf_{erb} + 0.5 \rfloor,$$

where

$$tf_{erb} = 21.4 \cdot \log_{10}\left(1 + 0.00437 \cdot tf \cdot {}^{2\pi}\!/_{f_s}\right).$$

8.C.3.4.2 Meixner amplitude parameters

This parameter is only present for Meixner type transients. Let *ta* hold the amplitude of the nth sinusoid under the Meixner transient in sub-frame sf of channel ch.

To convert to representation levels, the encoder quantizes on a logarithmic scale with a maximum amplitude error of 1.5dB. With ta_b=1.1885, the representation level $ta_{rl}[sf][ch][n]$ is calculated as follows

$$ta_{rl}[sf][ch][n] = \left\lfloor \frac{\log(ta)}{2 \cdot \log(ta_b)} + 0.5 \right\rfloor .$$

8.C.3.4.3 Meixner phase parameters

This parameter is only present for Meixner type transients. Let *tp* hold the phase of the nth sinusoid under the Meixner transient in sub-frame sf of channel ch. The phase is in the range of [-π, π>.

To convert to representation levels, the encoder quantizes on a linear scale with a maximum phase error tp_e of π/32 radians as defined in section 8.5.2. The representation level $tp_{rl}[sf][ch][n]$ is calculated as follows

$$tp_{rl}[sf][ch][n] = \left\lfloor \frac{tp}{2 \cdot tp_e} + 0.5 \right\rfloor ,$$

and tp_{rl} values equal to +16 are set to –16 (because –π is equal to π).

8.C.4 Sinusoids

The sinusoid set of encoder modules analyses the residual after transient subtraction for sinusoidal components. Up to 60 psycho-acoustically relevant sinusoids can be determined. This block consists of the following modules

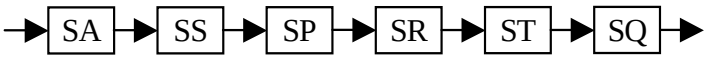


Figure 8.C.5 — Sinusoid processing blocks

To improve efficiency in coding the partials, the encoder tries to form tracks of closely related partials across sub-frames. The bits-stream formatter can encode these partials differentially across sub-frames yielding a lower bit rate.

The encoder has to keep track of the tracks and the components it is made up of. For this, the encoder has three additional data structures:

- A track index: keeps a record of the duration in sub-frames of a certain track. A birth is labeled with a one; continuations are labelled two and up.
- A track next pointer: given the current component, this number gives the index into the next sub-frame for the current track.
- A track previous pointer: given the current component, this number gives the index into the previous sub-frame for the current track.

8.C.4.1 Sinusoid analysis (SA)

The SA module determines frequencies using a high resolution FFT algorithm. This algorithm uses the FFT and the first derivative of the FFT to get an accurate estimate for the frequency of the sinusoidal components. The algorithm is reproduced below in pseudo MATLAB code. Let the variable x hold the input audio data buffer.

```
len = length(x);
x0 = hanning(len) .* x;
dx0 = hanning(len) .* (fs * diff(x));
dft0 = fft(x0);
dft1 = fft(dx0);

[tmp, idx] = max(dft0);
w = 2 * pi * (idx – 1) ./ len;
gain = w ./ (2 .* sin(w ./ 2));
fp = (gain ./ fs) .* (dft1(idx) ./ dft0(idx));
```

The variable fp now holds the frequency estimation for the highest bin in the FFT. For a detailed description and discussion of the algorithm used, see reference [FFT].

The module selects 1024 samples from the available audio data; the data gets centred on the middle of four sub-frames. It then finds every bin that has an amplitude peak relative to its neighbours. Because there are almost no relevant sinusoidal components above 10kHz, the module extracts only frequencies up to this limit. Finally, the high resolution FFT algorithm corrects the bin frequency. If the corrected frequency falls too far from the original bin, it is rejected.

After extracting frequencies for up to 60 partials, the module recalculates the Fourier transform of the data on the frequency estimate to obtain an amplitude estimate. The phase is estimated from the original bin and then corrected to minimize the error of the estimation.

8.C.4.2 Sinusoid synthesis (SS)

The sinusoid synthesis module synthesizes sinusoid components. The module uses a 50% overlap-and-add strategy. The sinusoid parameters of a sub-frame get synthesized over *L* samples, windowed by a Hanning window and added to the current synthesized signal. Thus each synthesized signal effectively consists of sinusoids from the current and previous sub-frames, allowing smooth transitions on the sub-frame boundaries. The residual passed on to the next module is the residual audio data after transient synthesis minus the synthesized signal from this module.

8.C.4.3 Sinusoid psycho (SP)

Psycho-acoustic modelling decreases the amount of information to be encoded in the bit stream by removing the sinusoids that are irrelevant for the human ear (as represented by the psycho-acoustic model). For example, we can remove one of two sinusoids that are close enough in frequency, if its amplitude is much lower, because it is imperceptible by the human auditory system. This phenomenon is known as “masking”.

The goal of this module is to determine the SMR for each sinusoidal component and to later eliminate ‘definitely imperceptible’ components in order to obtain a list of ‘perceptually relevant’ sinusoidal components. The algorithm used here is the same as in [HILN].

First, a psycho acoustical model is applied to the analysis interval, which results in a ‘masking curve’ for that analysis interval. Each candidate sinusoid is then compared with respect to this ‘masking curve’, resulting in a SMR for each sinusoidal component. This information is passed on for further processing.

8.C.4.4 Sinusoid remove (SR)

The sinusoid remove module takes the SMR of each partial and compares it to a certain threshold-parameter. If a component falls below the threshold, the component is removed from the data structure. The intended result is to eliminate ‘perceptually irrelevant’ sinusoids.

8.C.4.5 Sinusoid tracking (ST)

To improve the coding efficiency of sinusoidal components, this module tries to link sinusoidal components across sub-frames. This way, the frequency and amplitude parameters can be encoded differentially. To link

two components across sub-frames, compare frequency and amplitude of every component in the first sub-frame and try to match that to another component in the second sub-frame. The maximum allowed deviation for the frequency is 0.4 ERB and the maximum allowed deviation for the amplitude is a factor of three. The algorithm used here is the same as in [HILN].

8.C.4.6 Sinusoid quantisation (SQ)

The floating-point parameters must be quantized and coded before transmission. This module quantizes sinusoid parameters for frequency, amplitude and phase. The output of this module is a set of sinusoidal parameters that are suitable for the bit-stream format.

The granularity of the quantisation of the sinusoidal frequencies and amplitudes, denoted with freq_granularity and amp_granularity respectively, can be changed every frame. This influences continuing sinusoids that pass the frame boundary. As an example: assume a sinusoid with representation level 15 in sub-frame sf-1 (where amp_granularity = 0), which continues into sub-frame sf with representation level 8 and amp_granularity of 3 (see Figure 8.C.6). As the differences within this sub-frame sf are given using a amp_granularity of 3 it is not possible to represent the difference of –7, only –8 can be represented on this grid.

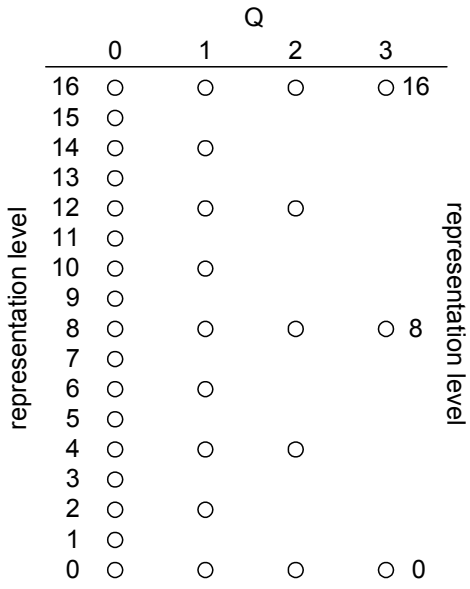


Figure 8.C.6 — Representation levels as function of quantisation grid

Prior to coding an amplitude difference, going from sub-frame sf-1 to sf, the previous amplitude representation level of sub-frame sf-1, is first converted to the granularity of the current sub-frame sf. So the representation level of the amplitude from which the difference will be coded will be first calculated as

$$\text{value\_prev\_sa} = 2^{\text{amp\_granularity}} \left\lfloor \frac{\text{value\_prev\_sa}}{2^{\text{amp\_granularity}}} + 0.5 \right\rfloor,$$

where value_prev_sa denotes the amplitude value of the previous sub-frame sf-1 and amp_granularity represents the granularity of the current sub-frame.

8.C.4.6.1 Amplitude quantization for births and continuations

Let *sa* hold the amplitude of the n^{th} sinusoid in sub-frame sf, channel ch. To convert to representation levels, the encoder quantizes the floating-point amplitudes on a logarithmic scale with a maximum amplitude error of 0.1875dB. With sa_b = 1.0218, the (integer) representation level $sa_{rl}[sf][ch][n]$ is calculated as follows

$$sa_{rl}[sf][ch][n] = \left\lfloor \frac{\log(sa)}{2 \cdot \log(sa_b)} + 0.5 \right\rfloor .$$

8.C.4.6.2 Frequency quantization for births

Let *f* hold the frequency of the nth sinusoid in sub-frame sf, channel ch. The frequency is expressed in radians. To convert to representation levels, the encoder quantizes the floating-point frequencies to an ERB-scale with 91.2 representation levels per ERB. The (integer) representation level $f_{rl}[sf][ch][n]$ is calculated as follows

$$f_{rl}[sf][ch][n] = \left\lfloor 91.2 \cdot \mathrm{erb}\left(\frac{2\pi f}{f_s} \right) + 0.5 \right\rfloor .$$

Where erb() is the function defined in section 8.C.3.4.1.

8.C.4.6.3 Phase quantization for births

Let *p* hold the phase of the nth sinusoid in sub-frame sf, channel ch. The phase is expressed in radians in the range [-π, π>. To convert to representation levels, the encoder quantizes the floating-point phases on a linear scale with a maximum phase error p_e of $\pi/32$ radians. The integer representation level $sp_{rl}[sf][ch][n]$ is calculated as follows

$$sp_{rl}[sf][ch][n] = \left\lfloor \frac{p}{2 \cdot p_e} + 0.5 \right\rfloor ,$$

and p_{rl} values equal to +16 are set to –16 (because –π is equal to π).

8.C.4.6.4 Frequency and phase quantization for continuations

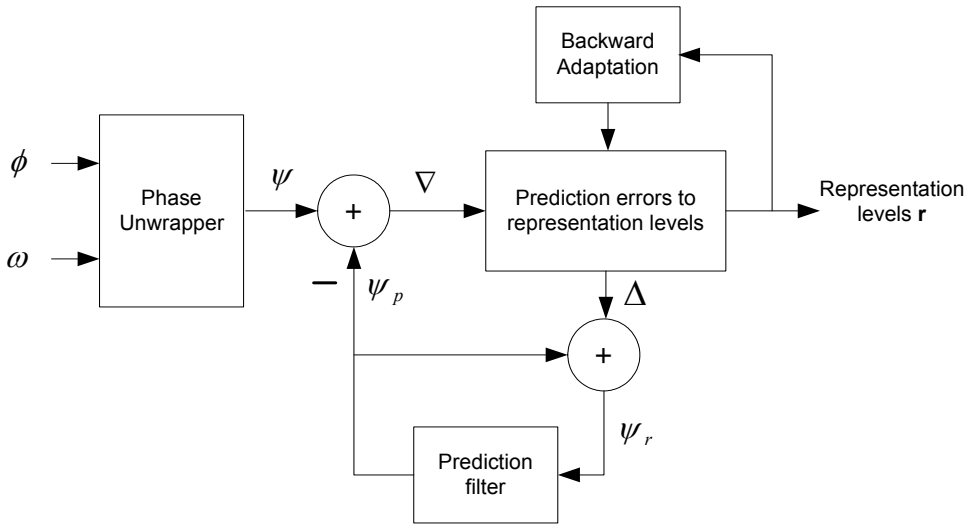


Figure 8.C.7 — The phase/frequency encoding system for continuations

The encoder structure is shown in Figure 8.C.7. The blocks 'Backward Adaptation' and 'Prediction Filter' are already present in the decoder and they have the same functionality. The two new blocks are the phase un-wrapper and the prediction error quantizer in block 'Phase Un-wrapper' and 'Prediction errors to representation levels' respectively.

Suppose we have a sinusoidal track of length κ, in frames *sf* = [*K, K+1, …, K+*κ *-1*] with frequencies *f*[*sf*][*ch*][*n*] (in radians) and phases *p*[*sf*][*ch*][*n*] (in radians) and the time distance between the sub frames which is represented by *S*. The phases are restricted by $-\pi \le p \le \pi$. The unwrapped phase is calculated by:

$$\psi[sf][ch][n] = p[sf][ch][n] + 2\pi \cdot m[sf][ch][n],$$

where for *sf = K+1, ..., K+*κ*-1*:

$$m[sf][ch][n] = e[sf][ch][n] + m[sf-1][ch][q],$$

where *q* represents the sinusoid index for sub-frame *sf*-1 and for *sf = K*:

$$m[K][ch][n] = 0.$$

The function *e* is the incremental unwrap factor that is defined as:

$$e[k] = round\left(\frac{f_{diff} \cdot S/2 - p_{diff}}{2\pi}\right),$$

with

$$f_{diff} = f[sf][ch][n] + f[sf-1][ch][q],$$

$$p_{diff} = p[sf][ch][n] - p[sf-1][ch][q].$$

The prediction error $\nabla[sf][ch][n]$ is calculated according to:

$$\nabla[sf][ch][n] = \psi[sf][ch][n] - \psi_p[sf][ch][n].$$

The quantisation is done as follows. The prediction error ∇ is compared to the boundaries *b*, such that the following equation is satisfied:

$$bl_i < \frac{\nabla[sf][ch][n]}{c[sf][ch][n]} \le bu_i.$$

From the value of *i*, that satisfies the above relation, the representation level s_delta_cont_freq_pha is set to the index in Table 8.C.1.

Table 8.C.1 — Quantisation table used for first continuation

Index *i*	Lower boundaries *bl*	Upper boundary *bu*
0	$-\infty$	-0.75*2
1	-0.75*2	0
2	0	0.75*2
3	0.75*2	∞

In order to minimize the additional delay in the decoder, the representation levels s_delta_cont_freq_pha are transmitted two sub-frames ahead. So, the representation level for sub-frame *sf* is transmitted in sub-frame *sf-2*. If the track is started in frame *sf-1*, the representation level of sub-frame *sf* will be transmitted in sub-frame *sf-1.* Therefore, a start of the track can contain up to 2 representation levels. In case of a refresh frame, the representation levels are transmitted in the same way as if the track would end in the last sub-frame before first sub-frame in the refresh-frame. The field refresh_sinusoids_next_frame is introduced for this purpose. For

the first sub-frame of a refresh frame, the index to Table 8.35 that corresponds to 0.75*c*[*sf*][*ch*][*n*] is used as s_adpcm_grid.

8.C.5 Noise

The noise is the last object to be analyzed by the SSC encoder. As illustrated in Figure 8.C.8, it consists of three modules.

Figure 8.C.8 — Noise processing blocks. NA = Noise analysis, NT = Noise temporal envelope and NQ = Noise quantization

8.C.5.1 Noise analysis (NA)

This module uses Laguerre modelling for noise parameterization. It takes the residual from the previous encoding stages as an input signal (see Figure 8.C.9). The Laguerre prediction filter is used as a means of modelling the noise colouring of the residual signal. A spectral estimation is performed, producing a whitened residual and a stream of quantized parameters.

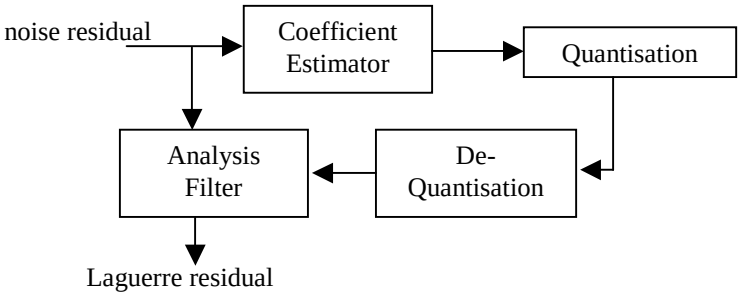


Figure 8.C.9 — Laguerre noise encoder

To ensure the stability of the algorithm, a high pass filter processes the signal to remove low frequency oscillations. The module then takes a segment of *L* samples of audio data, windows it with a Hanning window and estimates the optimal Laguerre prediction coefficients a_1, a_2, … a_K of the analysis filter AF (see Figure 8.C.10). It has been shown that for windowed input signals the AF is a minimum phase filter that performs whitening (reference [Laguerre1]). The prediction coefficients are calculated using the Levinson-Durbin algorithm.

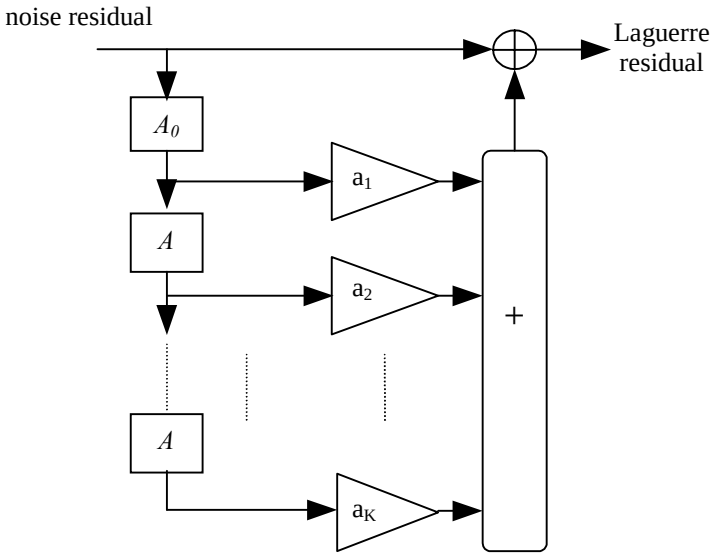


Figure 8.C.10 — Laguerre analysis filter

The filter A_0 is a first-order filter. The filters *A* are first-order all-pass sections.

$$A_0(z) = \sqrt{1-\lambda^2}\,\frac{z^{-1}}{1-z^{-1}\lambda}$$

$$A(z) = \frac{-\lambda + z^{-1}}{1-z^{-1}\lambda}$$

The Laguerre pole λ can be chosen such that the resulting frequency resolution of the analysis filter resembles that of the auditory system. The Laguerre coefficients are converted to the LAR coefficients that are written into the bit stream. First the Laguerre coefficients are calculated and then transformed to FIR coefficients (see 8.C.5.3.1). The normalized FIR coefficients are transformed to parcors (see 8.C.5.3.2) and then quantized by transforming them to LARs (Log Area Ratios) (see 8.C.5.3.3). Time differential encoding is employed. Conventional quantization is applied. Two different quantization scales are used (7 bit and 9 bit) and the applied scale is signalled to the decoder in n_laguerre_granularity. For more information on the algorithm, see reference [Laguerre2].

8.C.5.2 Noise Temporal Envelope (NT)

Using the inverse filtered (i.e. Laguerre filtered) noise residual, the temporal envelope is estimated (see Figure 8.C.11). This envelope is represented by 1 gain parameter and a number of LSF encoded LPC parameters per 4 sub-frames. The temporal envelope is typically estimated by applying an FFT onto the time signal and subsequently applying LPC analysis. The LPC coefficients *a* thus obtained, are transformed into LSF parameters *lsf*[*n*], for *n* = [*0, …,* n_nr_of_lsf-*1*], where n_nr_of_lsf is the LPC order. There are well known algorithms available to apply this conversion, see e.g. [Rothweiler].

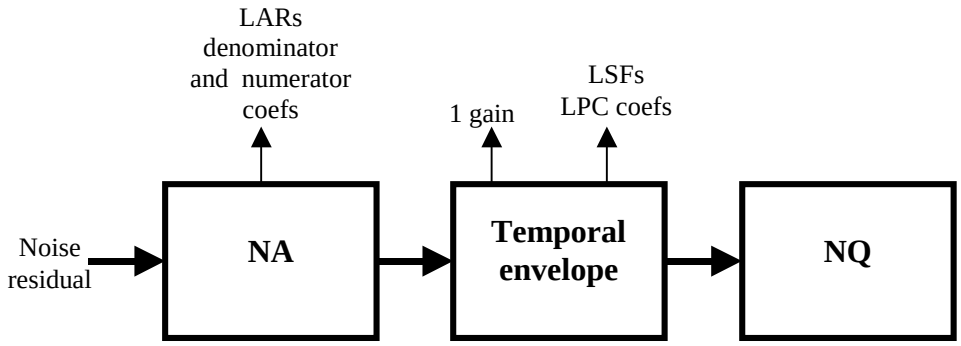


Figure 8.C.11 — Temporal envelope estimation

Gain parameter *G* is obtained by

$$G = \sqrt{\frac{\sum_i x[n]^2}{\sum_i e[n]^2}},$$

where x is the time signal on which the FFT is applied and e is the impulse response of filter $1/A(z)$. The filter $1/A(z)$ uses LPC coefficients *a*.

8.C.5.3 Noise quantisation (NQ)

This module quantizes the saved noise parameters to send to the bit-stream. The output of this module is a set of noise parameters that are suitable for the bit-stream format.

The filter parameters of numerator and denominator are first converted into LAR coefficients, because quantization of LAR coefficients influences the filter characteristic less than that of a- or reflection parameters.

8.C.5.3.1 Convert Laguerre coefficients into FIR coefficients

The FIR coefficients p' are generated from the Laguerre coefficients α by the following algorithm, where $\zeta = \sqrt{1-\lambda^2}$:

$$p_0 = 1 - \lambda\frac{\alpha_1}{\zeta},$$
$$p_k = -(\alpha_k + \lambda\alpha_{k+1})/\zeta, \text{ for } k = [1,..., \text{n\_nr\_of\_den} - 1],$$
$$p_k = -\alpha_k/\zeta,$$
$$p_k^{'} = p_k / p_0,$$

8.C.5.3.2 Convert FIR coefficients into parcors

The following algorithm in C code describes the conversion of m+1 FIR a-parameters p' into m (m=n_nrof_den) parcors rfc.

```
/* NOTE: p[0] must be 1 */
for (k = 0; k < m; k++)
{
    d[k] = –p'[k+1];
}
for (k = m-1; k >= 0; k–)
{
    rfc[k] = d[k];
    for (i = 0; i < k; i++)
    {
        tmp[i] = (d[i] + rfc[k] * d[k-i-1])/(1 - rfc[k]*rfc[k]);
    }
    for (i = 0; i < k; i++)
    {
        d[i] = tmp[i];
    }
    rfc[k] = –rfc[k];
}
```

8.C.5.3.3 Convert parcors into LAR coefficients

The following algorithm in C code describes the conversion of m parcors rfc into m LAR coefficients *lar*.

```
for (k = 0; k < m; k++)
{
    lar[k] = log_e((1+rfc[k])/(1-rfc[k]));
}
```

8.C.5.3.4 Quantisation of LAR coefficients

The floating point LAR coefficients typically have a range of [-8, +8]. LAR coefficients outside this range may be mapped onto the closest boundary (e.g. –9.1 is mapped onto -8). The quantisation step size is defined as Δ_{LAR} in section 8.6.3.3.1. The encoder determines the representation level $nlar_{rl}$ for a LAR coefficient $lar[k]$ as follows

$$nlar_{rl}[k] = \min\left(\max\left(\left\lfloor \frac{lar[k]}{\Delta_{LAR}} + 0.5 \right\rfloor, -\left(2^{lar\_bits-1} - 1\right) \right), 2^{lar\_bits-1} - 1 \right).$$

where *lar_bits*=9.

8.C.5.3.5 Quantisation of gains

Gains are quantised on a logarithmic scale using the following equation

$$ngain_{rl} = \begin{cases} 0 & G == 0 \\ \lfloor 20 \cdot \log_{10}(G) \rfloor + 21 & \text{otherwise} \end{cases}.$$

8.C.5.3.6 Quantization of LSF parameters

The LSF parameters are uniformly quantized using the equation

$$nlsf_{rl}[k] = \left\lfloor \frac{lsf[k]}{\pi} \cdot 255 + \frac{1}{2} \right\rfloor.$$

8.C.6 Parametric stereo

8.C.6.1 Mono signal generation

The monaural signal is generated according to

$$M = \frac{L+R}{2}.$$

8.C.6.2 Parameter estimation

In order to estimate the stereo parameters the signals M, L and R are analyzed using the hybrid filterbank as in Figure 8.20 and Figure 8.22 for the 10,20 bands and 34 bands configuration respectively. This results in the (sub-)subband domain signals $m(k,n)$, $l(k,n)$ and $r(k,n)$.

To estimate the parameters at sample position n' the following is calculated:

$$\begin{aligned}
e_l(b) &= \sum_{k=k_l}^{k_h}\sum_{n=0}^{L-1}\left| l\left(k, n'-\frac{L}{2}+1+n\right)\right|^2 + \varepsilon \\
e_r(b) &= \sum_{k=k_l}^{k_h}\sum_{n=0}^{L-1}\left| r\left(k, n'-\frac{L}{2}+1+n\right)\right|^2 + \varepsilon \\
e_R(b) &= \sum_{k=k_l}^{k_h}\sum_{n=0}^{L-1} l\left(k, n'-\frac{L}{2}+1+n\right) r^*\left(k, n'-\frac{L}{2}+1+n\right) + \varepsilon \\
e_O(b) &= \sum_{k=k_l}^{k_h}\sum_{n=0}^{L-1} l\left(k, n'-\frac{L}{2}+1+n\right) m^*\left(k, n'-\frac{L}{2}+1+n\right) + \varepsilon
\end{aligned},$$

where $e_l(b)$, $e_r(b)$ and $e_R(b)$ are the left channel excitation, the right channel excitation, the non-normalized cross-channel excitation between left and right channel and the non-normalized cross-channel excitation between left and mono channel for stereo bin b respectively, L the segment length, ε a very small value preventing division by zero ($\varepsilon = 1e-10$). The summation over k is shown in Table 8.C.2 and Table 8.C.3 for 20 bands and 34 bands respectively. In the case of 10 bands, Table 8.45 needs to be applied additionally.

Table 8.C.2 — Summation range in 71 sub subbands in case of 20 bands.

Parameter index *b*	sub subband index range *k*	QMF channel
0	2	0
1	3	0
2	4	0
3	5	0
4	6	1
5	7	1
6	8	2
7	9	2
8	10	3
9	11	4
10	12	5
11	13	6
12	14	7
13	15	8
14	16-17	9-10
15	18-20	11-13
16	21-24	14-17
17	25-29	18-22
18	30-41	23-34
19	42-70	35-63

Table 8.C.3 — Summation range in 91 sub subbands in case of 34 bands.

Parameter index *b*	sub subband index range *k*	QMF channel
0	0	0
1	1	0
2	2	0
3	3	0
4	4	0
5	5	0
6	16	1
7	17	1
8	18	1
9	19	1
10	20	2
11	21	2
12	26	3
13	27	3
14	28	4
15	29	4
16	32	5
17	33	6
18	34	7
19	35	8
20	36	9
21	37	10
22	38-39	11-12
23	40-41	13-14
24	42-43	15-16
25	44-45	17-18
26	46-47	19-20
27	48-50	21-23
28	51-53	24-26
29	54-56	27-29

30	57-59	30-32
31	60-63	33-36
32	64-67	37-40
33	68-90	41-63

The IID, denoted as $iid(b)$, the IPD, denoted as $ipd(b)$, the OPD, denoted as $opd(b)$, and the ICC, denoted as $\rho(b)$, for each stereo band b are calculated as:

$$iid(b) = 10\log_{10}\left(\frac{e_l(b)}{e_r(b)}\right)$$
$$ipd(b) = \angle e_R(b)$$
$$opd(b) = \angle e_O(b)$$
$$\rho(b) = \frac{|e_R(b)|}{\sqrt{e_l(b)e_r(b)}}$$

where $\angle$ represents the (four-quadrant) angle of a complex value.

Finally, these values are quantized (accordingly Table 8.25 or Table 8.26, Table 8.28 and Table 8.31) and coded into the bit-stream.

8.C.6.3 Quantization of IID

The IIDs are quantized according:

$$iid_{rl}(b) = \arg\left(\min\left|iid(b) - \mathbf{IID}\right|\right),$$

where **IID** represents the repertoire of IID values given in Table 8.25 and Table 8.26.

8.C.6.4 Quantization of ICC

The ICCs are quantized according:

$$icc_{rl}(b) = \arg\left(\min\left|icc(b) - \mathbf{ICC}\right|\right),$$

where **ICC** represents the repertoire of ICC values given in Table 8.28.

8.C.6.5 Quantization of IPD and OPD

The IPD and OPD values are quantized according to:

$$ipd_{rl}(b) = \arg\left(\min\left|\mathrm{mod}(ipd(b), 2\pi) - \mathbf{ANG}\right|\right) \quad ,$$
$$opd_{rl}(b) = \arg\left(\min\left|\mathrm{mod}(opd(b), 2\pi) - \mathbf{ANG}\right|\right) \quad ,$$

where $\mathbf{ANG} = \{0, \pi/4, 2\pi/4, 3\pi/4, \pi, 5\pi/4, 6\pi/4, 7\pi/4, 2\pi\}$. Finally, in case $ipd_{rl}(b) == 8$ it is set to $ipd_{rl}(b) = 0$. Likewise, in case $opd_{rl}(b) == 8$ it is set to $opd_{rl}(b) = 0$.

8.C.7 Bit-Stream Formatter

All processed sub-frames go into the bit-stream module. This module joins parameters of 8 subsequent sub-frames into one frame, applying loss-less coding (e.g. differences of parameters instead of their absolute value and Huffman coding) in order to save space. To enable random access to the soundtrack and to enable playback in case when one frame was lost during transmission, “refresh frames” can be sent (periodically) with absolute values of parameters.

8.C.7.1 Transients

8.C.7.1.1 Transient location

This parameter is coded into a number of bits, depending on *S*

$$\#bits = \lceil \log_2(S) \rceil .$$

8.C.7.1.2 Meixner sinusoids

Let $tf_{rl}[sf][ch][n]$ hold the frequency representation level of the n[th] sinusoid under the Meixner transient in sub-frame sf of channel ch. The coded version of this variable t_freq[sf][ch][n] is defined as follows

$$\text{t\_freq[sf][ch][n]} = tf_{rl}[sf][ch][n] .$$

Let $ta_{rl}[sf][ch][n]$ hold the amplitude representation level of the n[th] sinusoid under the Meixner transient in sub-frame sf of channel ch. The coded version of this variable t_amp[sf][ch][n] is defined as

$$\text{t\_amp[sf][ch][n]} = ta_{rl}[sf][ch][n] .$$

Let $tp_{rl}[sf][ch][n]$ hold the phase representation level of the n[th] sinusoid under the Meixner transient in sub-frame sf of channel ch. The coded version of this variable t_phi[sf][ch][n] is defined as

$$\text{t\_phi[sf][ch][n]} = tp_{rl}[sf][ch][n] .$$

8.C.7.2 Sinusoids

The bit-stream formatter exploits redundancy between births by coding the frequency and amplitude of the first birth in a sub-frame absolute and the rest of the births relative to the first one. It also exploits redundancy between successive parameters in a continuation; amplitude parameters are differentially coded between sub-frames. The Frequency and phase information is coded by an ADPCM quantizer. For the sinusoids in the first sub-frame of a frame, s_cont is determined as described in 8.6.2.1.

8.C.7.2.1 Absolute parameters for births

Let $f_{rl}[sf][ch][n]$ hold the representation level of the frequency of the n[th] sinusoid in sub-frame sf of channel ch and let *Qf* hold the granularity of the frequency quantisation grid. The frequency representation level is converted to s_freq[sf][ch][n] as follows

$$\text{s\_freq[sf][ch][n]} = \left\lfloor \frac{f_{rl}[sf][ch][n]}{2^{Qf}} + 0.5 \right\rfloor \cdot 2^{Qf} .$$

From this value s_freq_coarse[sf][ch][n] and s_freq_fine[sf][ch][n] are determined as:

$$\text{s\_freq\_coarse[sf][ch][n]} = \left\lfloor \frac{\text{s\_freq[sf][ch][n]}}{8} + 0.5 \right\rfloor \cdot 8 ,$$

$$\text{s\_freq\_fine[sf][ch][n]} = \left(\text{s\_freq[sf][ch][n]} - \text{s\_freq\_ coarse[sf][ch][n]}\right) / 2^{Qf}.$$

The value s_freq_coarse[sf][ch][n] is coded into the bit-stream using Table 8.B.9. The value s_freq_coarse[sf][ch][n] is directly inserted into the bit stream.

Let $a_{rl}[sf][ch][n]$ hold the representation level of the amplitude of the nth sinusoid in sub-frame sf of channel ch and let *Qa* hold the granularity of the amplitude quantisation grid. The amplitude representation level is converted into s_amp[sf][ch][n] as follows

$$\text{s\_amp[sf][ch][n]} = \left\lfloor \frac{a_{rl}[sf][ch][n]}{2^{Qa}} + 0.5 \right\rfloor \cdot 2^{Qa}.$$

From this value s_amp_coarse[sf][ch][n] and s_amp_fine[sf][ch][n] are determined as:

$$\text{s\_amp\_coarse[sf][ch][n]} = \left\lfloor \frac{\text{s\_amp[sf][ch][n]}}{8} + 0.5 \right\rfloor \cdot 8,$$

$$\text{s\_amp\_fine[sf][ch][n]} = \left(\text{s\_amp[sf][ch][n]} - \text{s\_amp\_coarse[sf][ch][n]}\right) / 2^{Qa}.$$

The value s_amp_coarse[sf][ch][n] is coded into the bit-stream using Table 8.B.2. The value s_amp_coarse[sf][ch][n] is directly inserted into the bit stream.

Let $p_{rl}[sf][ch][n]$ hold the representation level of the phase of the nth sinusoid in sub-frame sf of channel ch. This variable is directly coded in s_phi[sf][ch][n] according to

$$\text{s\_ phi[sf][ch][n]} = p_{rl}[sf][ch][n].$$

8.C.7.2.2 Absolute parameters for continuations

In case of a refresh frame, the parameters for continuations are coded using their absolute values. In addition, phase information is also encoded in the stream. Coding frequency and amplitude uses the same formulas as in section 8.C.7.2.1. Furthermore, for continuations, s_adpcm_grid is encoded into the bitstream using Table 8.B.1.

8.C.7.2.3 Differential parameters for births

Per sub-frame all births are sorted by frequency in ascending order, the lowest frequency birth is encoded as absolute parameter using s_freq[sf][ch][n]. The remaining births in the sub-frame are differentially coded with respect to the previous birth in this sub-frame. Phase is coded as absolute value.

Let $f_{rl}[sf][ch][n]$ hold the representation level of the birth frequency of the nth sinusoid in sub-frame sf, channel ch. The encoder determines the difference between the frequency representation level of nth and (n-1)th birth in the sub-frame and converts this into s_delta_birth_freq[sf][ch][n] as follows

$$\text{s\_ delta\_birth\_freq[sf][ch][n]} = f_{rl}[sf][ch][n] - f_{rl}[sf][ch][n-1].$$

Similar to the absolute frequency births this value is split into s_delta_birth_freq_fine[sf][ch][n] (directly coded) and s_delta_birth_freq_coarse[sf][ch][n], coded using Table 8.B.10.

Let $a_{rl}[sf][ch][n]$ hold the representation level of the birth amplitude of the nth sinusoid in sub-frame sf, channel ch. The encoder determines the difference between the amplitude representation level of nth and (n-1)th birth in the sub-frame and converts this into s_delta_birth_amp[sf][ch][n] as follows

$$\text{s\_delta\_birth\_ amp[sf][ch][n]} = a_{rl}[sf][ch][n] - a_{rl}[sf][ch][n-1].$$

Similar to the absolute amplitude births this value is split into s_delta_birth_amp_fine[sf][ch][n] (directly coded) and s_delta_birth_amp_coarse[sf][ch][n], coded using Table 8.B.3.

8.C.7.2.4 Differential parameters for continuations

Let $a_{rl}[sf][ch][n]$ hold the representation level of the continuation amplitude of the nth sinusoid in sub-frame sf, channel ch. Let *track_prev*(*sf, ch, n*) hold the index into the previous sub-frame for $a_{rl}[sf][ch][n]$. The encoder determines the difference between the amplitude index of current and previous sub-frame and codes that using Table 8.B.5, Table 8.B.6, Table 8.B.7 or Table 8.B.8 depending on the amplitude granularity of the current frame in s_delta_birth_amp[sf][ch][n] as follows

$$\text{s\_delta\_cont\_amp[sf][ch][n]} = a_{rl}[sf][ch][n] - a_{rl}[sf-1][ch][track\_prev(sf,ch,n)].$$

For frequencies and phases s_delta_cont_freq_pha[sf][ch][n] are directly inserted into the bitstream.

8.C.7.3 Noise

8.C.7.3.1 Parameter λ

Two bits are used for the parameter λ of the Laguerre noise analysis filter - n_laguerre[ch]. One additional bit – n_laguerre_granularity[sf][ch] - is used for signalling the quantization step for the Laguerre parameters.

8.C.7.3.2 Absolute lar parameters for noise

The encoder directly codes the representation levels $nlar_{rl}$ into the bit-stream

$$\text{n\_lar\_den\_coarse[sf][ch][n]} = 4 \cdot \lfloor nlar_{rl}[n]/4 \rfloor.$$

Huffman Table 8.B.12 is used to encode n_lar_den_coarse. If n_laguerre_granularity == %1, then:

$$\text{n\_lar\_den\_fine[sf][ch][n]} = nlar_{rl}[n] - \text{n\_lar\_den\_coarse[sf][ch][n]}.$$

Then n_lar_den_fine is inserted in the bitstream.

8.C.7.3.3 Differential lar parameters for noise

The encoder determines the difference between the lar coefficient of the current and previous sub-frame and codes that directly in *n_delta_lar_den*[*sf*][*ch*][*n*] as follows

$$\text{n\_delta\_lar\_den\_coarse[sf][ch][n]} = 4 \cdot \lfloor (nlar_{rl}[n] - nlar_{rl,psf}[n])/4 \rfloor$$

where $nlar_{rl,psf}$ is the representation level of the previous sub-frame. If n_laguerre_granularity == %1, then:

$$\text{n\_delta\_lar\_den\_fine[sf][ch][n]} = (nlar_{rl}[n] - nlar_{rl,psf}[n]) - \text{n\_delta\_lar\_den\_coarse[sf][ch][n]}.$$

8.C.7.3.4 Noise gain parameters

The first gain of a refresh frame is coded by their absolute value.

$$\text{n\_gain[sf][ch]} = ngain_{rl}.$$

All following noise gains are encoded differentially.

$$\text{n\_delta\_gain[sf][ch]} = ngain_{rl} - ngain_{rl,p4sf},$$

where $ngain_{rl,p4sf}$ represents the gain representation level for sub-frame sf-4.

For i=0, n_lsf[sf][ch][0] = $nlsf_{rl}$[0]. For i > 0, n_delta_lsf[sf][ch][i] = $nlsf_{rl}$[i] - $nlsf_{rl}$[i-1]. Then n_lsf and n_delta_lsf are Huffman encoded using Table 8.B.13. If the representation levels $nlsf_{rl}$ of the current frame are similar to the ones of the previous frame, the current ones can be substituted by the previous ones. This is indicated by bit n_overlap_lsf (see 8.6.3.2.1).

8.C.7.4 Parametric stereo parameters

Either time or frequency differential coding is applied to the IID, IPD, OPD and ICC parameters. For all parameters the same type of coding applies. As an example the coding for IIDs is given. In case of frequency differential coding the indices are determined as:

$$\text{iid\_par\_df[e][0]} = iid_{rl}(0)$$
$$\text{iid\_par\_df[e][b]} = iid_{rl}(b) - iid_{rl}(b-1) \qquad \text{for} \, b > 0$$

In case of time differential coding the indices are determined as:

$$\text{iid\_par\_dt[e][b]} = iid_{rl}(b) - iid_{rl,pse}(b)$$

where $iid_{rl,pse}(b)$ represents the b-th IID representation level of the previous stereo envelope. In case no previous envelope is available the indices are determined as:

$$\text{iid\_par\_dt[e][b]} = iid_{rl}(b) - iid_{rl,pse}(b)$$

For IPD and OPD parameters differences outside the range are mapped into the existing range using the modulo function. Huffman coding of the stereo parameters is performed using Table 8.B.17, Table 8.B.18, Table 8.B.19, Table 8.B.20 and Table 8.B.21.

8.C.8 Tempo and pitch scaling in decoder

The goal of tempo scaling an audio signal is to alter the (instantaneous) duration of a signal while preserving the perceived pitch. Typically, in the decoder tempo scaling is handled for each object separately:

- Transients; tempo scaling is typically not applied to the transient object. Perceptually transients are characterised by the attack and decay times. A much more natural result is obtained when the length of a transient is preserved.

- Sinusoids; for the sinusoidal object tempo scaling can be obtained by applying two changes in the decoder. First of all the synthesis window length should be changed according to the time scaling factor ensuring unitary overlap-add. Secondly, a new phase value is obtained for continuations by applying the continuous phase function (see 8.C.8.1).

- Noise; within the noise object the tempo can be scaled by scaling both the synthesis window as well as the temporal envelope.

- Stereo: tempo scaling is done by changing the position of the parameter position according to the time scaling factor (see 8.C.8.2).

As the pitch is mostly determined by the sinusoidal frequencies one approach is to scale only the frequencies of the sinusoids as $f_{new} = p \cdot f_{old}$ with p the pitch scaling factor. In the case that the pitch scaled frequency f_{new} crosses the available bandwidth, the corresponding sinusoid is not synthesized. Notice that this also affects the continuous phase procedure.

8.C.8.1 Continuous phase

Since the phase values depend on the sub-frame size S, in case of tempo and pitch-scaling, the decoder switches to continuous phase. The tempo and pitch-scaling procedure for sinusoidal components is done as follows. First, the original frequency and phase are obtained by the decoding process as described in 8.6.2.2. In the case phase_jitter_present is set to %1, the obtained frequency is re-quantised as described in 8.C.4.6.2, resulting in representation level sf_{rl}. The frequency representation level sf_{rl} becomes sf_{rl} + sf_{jitter} and is de-quantised (see 8.5.2). This frequency of multiplied by a pitch scale factor and used in the continuous phase calculation.

The continuous phase φ_2 is calculated such that the phase of two consecutive segments matches in the middle of the overlapping region. In the example of Figure 8.C.12, the continuous phase φ_2 (and thus sp_q) is calculated as

$$\varphi_2 = \varphi_1 + n_1 \cdot f_1 + n_2 \cdot f_2, \quad n_1 = \frac{L}{4}, \quad n_2 = \frac{L}{4}.$$

Again note that the phase information φ_1 and φ_2 is defined for the middle of the segment (=(L-1)/2).Here f_1 and f_2 represent the frequency in radians.

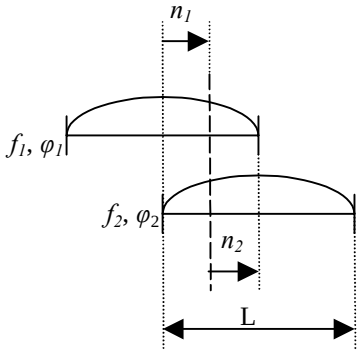


Figure 8.C.12 — Derivation of continuous phase information. The phase φ_2 is obtained from the parameters f_1, f_2 and φ_1

The phase of a continuing sinusoidal component in the first sub-frame of a frame with refresh_sinusoids==%1 is NOT set to the value read from the stream (s_phi), but calculated from the expression for φ_2. If the decoding process starts at this sub-frame, e.g. at random access from the noted sub-frame, the phase of a continuing sinusoidal component should be set to the value read from the stream.

8.C.8.2 Tempo scaling for stereo

Tempo scaling for the parametric stereo tool can be achieved by adapting the position of the parameters. Let μ be the tempo scale factor. Consider the previous and current (not necessarily integer) scaled parameter positions $\hat{n}_{prev} = n_{e-1} \cdot \mu$ and $\hat{n}_{curr} = n_e \cdot \mu$ respectively. By moving these positions to integer positions, the stereo decoding can be performed as usual. A method to map the non-integer parameter positions $\hat{n}_{prev}$ and $\hat{n}_{curr}$ to integer positions n_{prev} and n_{curr} is given using the following recursion.

$$m = \mathrm{mod}\left(\hat{n}_{curr} - n_{prev}, 1\right)$$

where n_{prev} is the previous integer position.

Then the current integer parameter position is calculated as follows:

$$n_{curr} = \hat{n}_{curr} + 1 - m$$

In order to initiate the recursion, $n_{prev} = 0$.

Contents for Subpart 9

Subpart 9: MPEG-1/2 Audio in MPEG-4

9.1 Scope

The MPEG-1/2 Audio in MPEG-4 subpart of MPEG-4 Audio specifies the usage of MPEG-1/2 Layer-1, 2 or 3 in an MPEG-4 oriented way, i.e. such that signalling and access unit handling on Systems level is identical to the other MPEG-4 audio object types.

In order to be carried in MPEG-4, the MPEG-1/2 Layer 1, 2 or 3 bitstream frames are re-formatted such that they become self-contained MPEG-4 access units. This facilitates transport over packet based networks, random access, and editability. Those self-contained access units, as used in an MPEG-4 Systems compliant transport or storage format, can be re-converted to MPEG-1/2 compliant bitstreames, and then decoded with any MPEG-1/2 compliant decoder. Several methods of re-conversion are given in an informative annex.

The MPEG-4 Audio syntax is further extended to allow multi-channel configurations based on ISO/IEC 11172-3 and ISO/IEC 13818-3 Layer 3. The multi-channel configurations are similar to the configurations defined for the other multi-channel capable MPEG-4 audio object types. Note that for MPEG-1/2 Layer 1 and 2 the format is not extended. The multi-channel format for these layers is described in ISO/IEC 13818-3.

Furthermore, the permitted sampling frequencies for Layer-3 are extended.

Assistance is furthermore provided in the use of decSpecificInfo and accessUnit as to utilize MPEG-1/2 Layer 1, 2 or 3 in the MPEG-4 world by means of the legacy MPEG-4 Systems interface using ObjectTypeIndication 0x69 or 0x6b.

9.2 MPEG_1_2_SpecificConfig

Table 9.1 — MPEG_1_2_SpecificConfig()

Syntax	No. of bits	Mnemonic
MPEG_1_2_SpecificConfig()		
{		
extension;	**1**	**bslbf**
}		

extension shall be zero.

9.3 Channel Mapping

The following rules apply:

- single_channel_element()'s and lfe_element()'s are represented by mono audio frames.
- channel_pair_element()'s are represented by stereo audio frames.
- For Layer-1 and Layer-2, not more than one mono audio frame representing a single_channel_element() or one stereo audio frame representing a channel_pair_element() is permitted.

9.4 Access Unit Format

9.4.1 Layer 1 and 2

One audio frame maps directly to one access unit.

9.4.2 Layer 3

One access unit consists of one or more mp3_channel_elements. An mp3_channel_element equals a Layer 3 audio frame with the following modifications compared to its definition in ISO/IEC 11172-3 or ISO/IEC 13818-3:

syncword (12 bit) — signals the total length in bytes of the mp3_channel_element (consisting of header, error_check, side info and main data).

main_data_begin (9/8 bit) — is either set to the correct value of the corresponding MPEG-1/2 Layer 3 bitstream or to zero.

main_data() — is generally stored after the side information.

All other data elements shall be set according to their specification in ISO/IEC 11172-3 or ISO/IEC 13818-3. All settings in the header shall correspond to the settings in the AudioSpecificConfig().

All mp3_channel_elements belonging to the same timestamp are stored sequentially into one access unit according to the order given in subpart 1, subclause 1.6.3.4, Table 1.11 (Channel Configuration). An example of a 5.1 channel configuration is given in Figure 9.1.

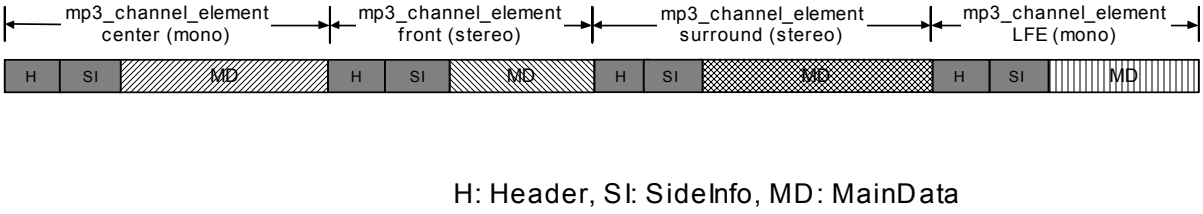


Figure 9.1 — Access Unit containing mp3_channel_elements for a 5.1 channel configuration

9.5 Sampling rate extension for Layer 3

This subclause provides specifications to allow the usage of Layer 3 with sampling rates not specified in ISO/IEC 11172-3 or ISO/IEC 13818-3.

The bitstream syntax and description for the extension towards sampling frequencies lower than those specified in ISO/IEC 13818-3 are in accordance of ISO/IEC 13818-3 (one frame covers 576 samples).

The subsequent subclauses outline the necessary extensions.

9.5.1 Bitrates

Table 9.1 specifies the bitrate depending on the bitrate_index and sampling frequency.

Table 9.1 — bitrate depending on bitrate_index and sampling frequency

bitrate_index	bitrate specified (kbit/s)		
	8, 11.025, 12 kHz	16, 22.05, 24 kHz (see ISO/IEC 13818-3)	32, 44.1, 48 kHz (see ISO/IEC 11172-3)
'0000'	free	free	free
'0001'	8	8	32
'0010'	16	16	40
'0011'	24	24	48
'0100'	32	32	56
'0101'	40	40	64
'0110'	48	48	80
'0111'	56	56	96
'1000'	64	64	112
'1001'	forbidden	80	128
'1010'	forbidden	96	160
'1011'	forbidden	112	192
'1100'	forbidden	128	224
'1101'	forbidden	144	256
'1110'	forbidden	160	320
'1111'	forbidden	forbidden	forbidden

9.5.2 Sampling frequency

Depending on the sampling frequency signaled in the AudioSpecificConfig, the data element sampling_frequency in the header has to be set as specified in Table 9.2.

Table 9.2 — Setting of the data element sampling_frequency depending on the sampling frequency specified in the AudioSpecificconfig()

sampling_frequency	sampling frequency
00	11.025 kHz and multiples thereof
01	12 kHz and multiples thereof
10	8 kHz and multiples thereof
11	reserved

9.5.3 Padding

Padding is necessary with a sampling frequency of 11.025 kHz and multiples thereof.

9.5.4 Scalefactor bands

The subdivision of the spectrum into scalefactor bands is fixed for every block length and sampling frequency and stored in tables in the coder and decoder. The tables for the sampling frequencies not specified in ISO/IEC 11172-3:1993 or ISO/IEC 13818-3:1998 are specified in Annex 9.A. In accordance with ISO/IEC 11172-3 or ISO/IEC 13818-3, the scale factor for frequency lines above the highest line in the tables is zero, which means that the actual multiplication factor is 1.0.

9.5.5 Intensity stereo mode

Step 3 of the intensity stereo mode decoding (see ISO/IEC 11172-3, subclause 2.4.3.4.9.3) is clarified as follows:

- if only the uppermost scalefactor band is in intensity stereo mode, then

  is_ratio(20) = 1 for long blocks

  is_ratio(11) = 1 for short blocks

- if at least the upper two scalefactor bands are in intensity stereo mode, then

is_ratio(20) = is_ratio(19) for long blocks

is_ratio(11) = is_ratio(10) for short blocks

Annex 9.A
(normative)

Scalefactor band tables

Table 9.A.1 — 8 kHz sampling rate, long blocks, number of lines 576

scalefactor band	width of band	index_of_start	index_of_end
0	12	0	11
1	12	12	23
2	12	24	35
3	12	36	47
4	12	48	59
5	12	60	71
6	16	72	87
7	20	88	107
8	24	108	131
9	28	132	159
10	32	160	191
11	40	192	231
12	48	232	279
13	56	280	335
14	64	336	399
15	76	400	475
16	90	476	565
17	2	566	567
18	2	568	569
19	2	570	571
20	2	572	573

Table 9.A.2 — 8 kHz sampling rate, short blocks, number of lines 192

scalefactor band	width of band	index_of_start	index_of_end
0	8	0	7
1	8	8	15
2	8	16	23
3	12	24	35
4	16	36	51
5	20	52	71
6	24	72	95
7	28	96	123
8	36	124	159
9	2	160	161
10	2	162	163
11	2	164	165

Table 9.A.3 — 11.025 kHz sampling rate, long blocks, number of lines 576

scalefactor band	width of band	index_of_start	index_of_end
0	6	0	5
1	6	6	11
2	6	12	17
3	6	18	23
4	6	24	29
5	6	30	35
6	8	36	43
7	10	44	53
8	12	54	65
9	14	66	79
10	16	80	95
11	20	96	115
12	24	116	139
13	28	140	167
14	32	168	199
15	38	200	237
16	46	238	283
17	52	284	335
18	60	336	395
19	68	396	463
20	58	464	521

Table 9.A.4 — 11.025 kHz sampling rate, short blocks, number of lines 192

scalefactor band	width of band	index_of_start	index_of_end
0	4	0	3
1	4	4	7
2	4	8	11
3	6	12	17
4	8	18	25
5	10	26	35
6	12	36	47
7	14	48	61
8	18	62	79
9	24	80	103
10	30	104	133
11	40	134	173

Table 9.A.5 — 12 kHz sampling rate, long blocks, number of lines 576

scalefactor band	width of band	index_of_start	index_of_end
0	6	0	5
1	6	6	11
2	6	12	17
3	6	18	23
4	6	24	29
5	6	30	35
6	8	36	43
7	10	44	53
8	12	54	65
9	14	66	79
10	16	80	95
11	20	96	115
12	24	116	139
13	28	140	167
14	32	168	199
15	38	200	237
16	46	238	283
17	52	284	335
18	60	336	395
19	68	396	463
20	58	464	521

Table 9.A.6 — 12 kHz sampling rate, short blocks, number of lines 192

scalefactor band	width of band	index_of_start	index_of_end
0	4	0	3
1	4	4	7
2	4	8	11
3	6	12	17
4	8	18	25
5	10	26	35
6	12	36	47
7	14	48	61
8	18	62	79
9	24	80	103
10	30	104	133
11	40	134	173

Annex 9.B
(informative)

Converting MPEG-1/2 Layer 3 bitstreams into mp3_channel_elements

The use of the bit-reservoir usually causes the start of the main_data() to appear in a past bitstream frame. This needs to be modified by moving the main_data() immediately adjacent to its side information. This is illustrated for one Layer 3 bitstream (mono or stereo) in Figure 9.B.1. Each resulting mp3_channel_element is mapped directly into an access unit. Resulting header and side info are indicated by H' and SI' respectively.

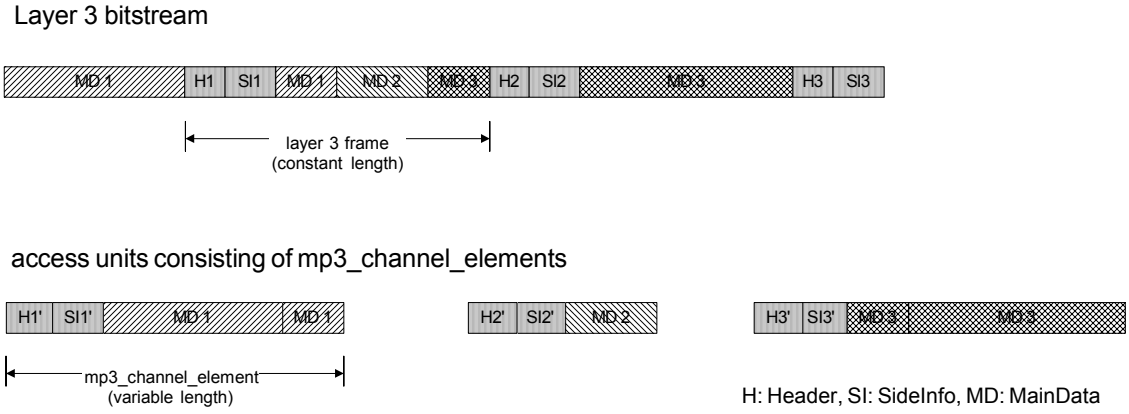


Figure 9.B.1 — Converting an MPEG-1/2 Layer 3 bitstream into mp3_channel_elements

All data elements of the header() shall be preserved. The data element main_data_begin might be set to zero. In that case the CRC has to be recalculated.

Annex 9.C
(informative)

Converting mp3_channel_elements into MPEG-1/2 Layer 3 bitstreams

9.C.1 Overview

mp3_channel_elements extracted from an access unit have to undergo the following conversion operations in order to obtain MPEG-1/2 Layer 3 audio bitstreams compliant to ISO/IEC 11172-3 or ISO/IEC 13818-3:

- for each mp3_channel_element per access unit open a decoder instance or output stream
- for each mp3_channel_element in every access unit do
- restore syncword and IDex
- correct bitrate_index
- adjust main_data_begin
- recalculate crc_word
- reconstruct framing

9.C.2 Sampling rate signaling

The last bit of the syncword shall be used in a backwards compatible way to allow the signalling of sampling rates not specified in ISO/IEC 11172-3 or ISO/IEC 13818-3. This leads to the following syntax modification:

Syntax	No. of bits	Mnemic
header()		
{		
syncword;	**11**	**bslbf**
Idex;	**1**	**bslbf**
…		

syncword The bit string '1111 1111 111'.

IDex One bit to indicate the extended ID of the algorithm. Has the value '0' for sampling rates not specified in ISO/IEC 11172-3 or ISO/IEC 13818-3.

The following table specifies the sampling rate depending on the values for IDex and ID:

IDex	ID	sampling rate
0	0	8, 11.025, 12 kHz
1	0	16, 22.05, 24 kHz (see ISO/IEC 13818-3)
1	1	32, 44.1, 48 kHz (see ISO/IEC 11172-3)

9.C.3 Reconstruction instructions

This reconstruction process offers some degrees of freedom:

bitrate_index (stuffing might be required to adjust the bitstream frame length according to the new bitrate_index, sampling frequency and padding_bit settings)

1) set to the maximum value allowed (signalling the maximum allowed bitstream frame length).

2) set to nearest higher value that matches the mp3_channel_element length.

3) set to nearest higher value that matches the mp3_channel_element length minus main_data_begin of the current audio frame.

main_data_begin

4) set to zero.

5) set to the value pointing back to the end of main_data of the previous audio frame.

6) set to the correct value of the corresponding MPEG-1/2 Layer 3 bitstream.

location of stuffing

7) at end of main_data: preserves ancillary data written in forward direction, starting after the last Huffman code word.

8) at the end of the last Huffman codeword (location can be calculated using the part_2_3_length): preserves ancillary data written in reverse direction starting before the main_data of the next frame.

9) no stuffing required: preserves any ancillary data.

Depending on bitrate requirements and ancillary data handling, these possibilities can be combined in several ways:

The simplest method sets the bitrate to the maximum size. This is the preferred method when feeding existing MPEG-1/2 Layer 3 decoders. main_data_begin is set to zero. Stuffing bits are added either before or after ancillary data (see Figure 9.C.1, example A and B).

A more advanced method can be derived from this simple method by setting the bitrate_index to the nearest higher value that corresponds to the length of the mp3_channel_element. With this modification, the bitrate can be significantly reduced (see Figure 9.C.1, example C and D).

To avoid the necessity of stuffing, main_data_begin is set to the value pointing back to the end of main_data of the previous frame. The bitrate_index is now set to the nearest higher value that matches the mp3_channel_element length minus main_data_begin of the current audio frame (see Figure 9.C.1, example E in Figure 9.C.1). Only if main_data_begin would exceed the allowed value, stuffing has to be performed.

The original Layer 3 bitstream can be reconstructed perfectly, if the correct main_data_begin value of the corresponding MPEG-1/2 Layer 3 bitstream was preserved (see Figure 9.C.1, example F).

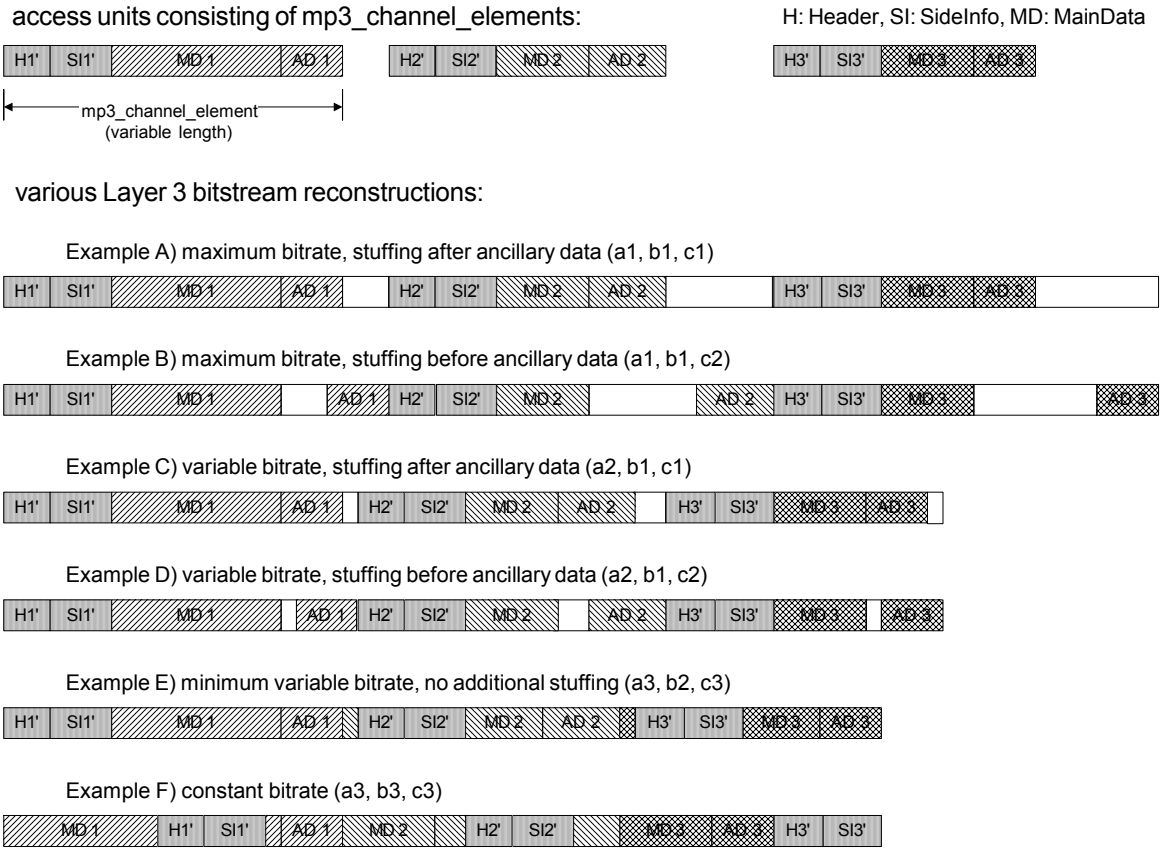


Figure 9.C.1 — Converting mp3_channel_elements to MPEG-1/2 Layer 3 bitstreams

Annex 9.D
(informative)

Legacy MPEG-4 Systems Interface to MPEG-1/2 Audio

9.D.1 Overview

This annex provides assistance in the use of decSpecificInfo and accessUnit to utilize MPEG-1/2 Layer 1, 2, 3 and MPEG-2 AAC in the MPEG-4 world using the following objectTypeIndication values:

- 0x6b (Audio ISO/IEC 11172-3)
- 0x69 (Audio ISO/IEC 13818-3)
- 0x66 (Audio ISO/IEC 13818-7 Main Profile)
- 0x67 (Audio ISO/IEC 13818-7 Low Complexity Profile)
- 0x68 (Audio ISO/IEC 13818-7 Scalable Sampling Rate Profile)

9.D.2 Decoder Specific Information

In ISO/IEC 14496-1 the **decSpecificInfo** is defined for certain media decoder information. This decoder specific information constitutes an opaque container with information for a specific media decoder. If available, it can be used for decoder initialization and a-priory instantiation of the compositor. It is not necessary to define this decoder specific information, its existence and semantics depend on the values of DecoderConfigDescriptor.streamType and DecoderConfigDescriptor.objectTypeIndication.

The lack of the availability of any decSpecificInfo leads to the situation that the format of the Composition Memory cannot be distinguished a-priory to instantiate the compositor. Hence, the decoder specifies the format of the composition memory.

9.D.2.1 MPEG-2 AAC

For MPEG-2 AAC a decSpecificInfo is defined, i.e. in the case of DecoderConfigDescriptor.objectTypeIndication values that refer to streams complying with ISO/IEC 13818-7 (MPEG-2 AAC).

In this case audio decoders receive all relevant information from that decSpecificInfo, which consists of an adif_header(), and can forward the composition memory format to the composition memory.

9.D.2.2 MPEG-1 Audio and MPEG-2 Audio

No decSpecificInfo is defined for MPEG-1 Audio or MPEG-2 Audio, i.e. in the case of DecoderConfigDescriptor.objectTypeIndication values that refer to streams complying with ISO/IEC 11172-3 (MPEG-1 Audio) and ISO/IEC 13818-3 (MPEG-2 Audio). In these cases audio decoders receive all relevant information in the ‘header()’ element of their own bitstream and can forward the composition memory format to the composition memory. Thus, changes in the output format need to be dealt with dynamically, i.e. without an elementary stream descriptor update.

9.D.3 Access Units

An MPEG-1/2 Layer 1, 2 or 3 frame (the data between sync words) or MPEG-2 AAC frame (raw_data_block()) can be treated as an audio access unit not only in the context of ISO/IEC 11172 and ISO/IEC 13818 but also in the context of ISO/IEC 14496.

When treating MPEG-1/2 Layer 1, 2, 3 or MPEG-2 AAC frames as MPEG-4 units, timing information is usually assigned to such access units.

Since the definitions of an audio access unit do not match exactly between MPEG-1/2 on one hand and MPEG-4 on the other hand, some special considerations have to be taken into account.

Particularly for Layer 3, an audio access unit is specified in MPEG-1/2 as a part of the bitstream that might be decodable only with the use of previously acquired main information, which does not reflect the definition of an audio access unit in MPEG-4.

Subsequently, some audio access units might not be decodable due to the lack of some lost main information in the case of bitstream punching and random access. However, the timing information is kept correctly.

In the case it is deemed necessary to have better editing or punch-in abilities for Layer 3 streams, it is advisable to use VBR encoded streams. It is possible to convert any existent Layer 3 stream into a VBR stream

- Unambiguously
- Fully MPEG-1 or MPEG-2 compliant
- Decodable by any existing Layer 1, 2 or 3 decoder

This can be done as follows:

The main_data() for a single frame is put immediately adjacent to its side information. The main_data_begin pointer is set to zero. The frame by frame bitrate indices (bitrate_index) are increased to the minimum value needed to obtain a frame-length that can accommodate the original header, error_check, side info and main data. Due to the granularity in the available bitrates, generally this frame length is larger than the length of the header, error_check, side info and main data. In this case stuffing bits are added at the end of main_data to obtain compliant frames.

Contents for Subpart 10

Subpart 10: Technical description of lossless coding of oversampled audio

10.1 Scope

This subpart of ISO/IEC 14496-3 describes the MPEG-4 lossless coding algorithm for oversampled audio signals.

10.2 Terms and definitions

The following definitions are used in this document.

Audio Channel	The stream of DSD bits intended for one loudspeaker.
Audio Frame	A Frame containing Audio data.
Audio Channel Number	The sequence number assigned to an Audio Channel. Audio Channel Numbers are contiguously numbered starting with one.
Frame	A block of data belonging to a certain Time Code. The playing time of a Frame is 1/75 Sec.
Reserved	All fields labelled Reserved are reserved for future standardization. All Reserved fields must be set to zero.
Silence Pattern	A digitally generated DSD pattern with the following properties:

- All Audio Bytes (see 10.6.1.1) have the same value.
- Each Audio Byte must contain 4 bits equal to zero and 4 bits equal to one.

Direct Stream Digital	A one bit oversampled representation of the audio signal.
Direct Stream Transfer	The lossless coding technique used for DSD signals in Super Audio CD.
DSD	See Direct Stream Digital.
DST	See Direct Stream Transfer.
Half Probability	Half Probability defines for each Audio Channel in an Audio Frame whether the first DSD bits are arithmetically encoded using the Ptable values, or using a probability equal to ½.
Mapping	Mapping defines, for each Segment, the Prediction Filter and Probability Table that is used.
Prediction Filter	A Prediction Filter is a transversal filter used to predict the value of the next DSD bit. A Prediction Filter is characterized by a prediction order and by coefficients.
Probability Table	A Probability Table contains the probability that the value of a DSD bit is predicted erroneously for a given output of the prediction filter.
Ptable	See Probability Table.

Sampling Frequency	The sampling frequency of the DSD signal shall be 64 * 44.1 kHz, 128 * 44.1 kHz or 256 * 44.1 kHz.
Segmentation	Each Audio Channel in an Audio Frame can be partitioned into Segments.

10.3 Conventions

In this document the conventions as described in this subclause are used.

10.3.1 Arithmetic and bit operations

a>>b	Right shift a over b bits. The new msb bits are set to '0'.
a<<b	Left shift a over b bits. The new lsb bits are set to '0'.
a | b	Bitwise OR of a and b.
a & b	Bitwise AND of a and b.
min(a,b)	Minimum value of a and b.
max(a,b)	Maximum value of a and b.
a mod b	Value of a modulo b.
trunc(a)	Value of a, rounded downwards.
| a |	Absolute value of a.
a==b	Evaluate if a is equal to b.
a!=b	Evaluate if a is not equal to b.
a=b	Variable a is set to the value of b.
a++	a = a + 1.
a -= b	a = a – b.
a += b	a = a + b.

10.3.2 Bit ordering

The graphical representation of all multiple-bit quantities is such that the most significant bit (msb) is on the left, and the least significant bit (lsb) is on the right. Figure 10.1 defines the bit position in a Byte.

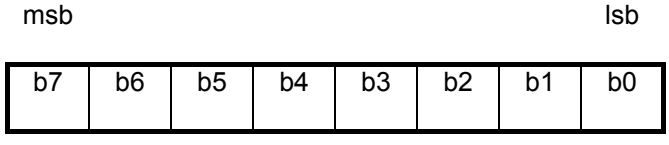


Figure 10.1 — Bit ordering in a Byte

10.3.3 Bit sequence

In all places where a bit sequence is used, a most significant bit first notation is used.

10.3.4 Decimal notation

All Decimal values are preceded by a blank space or the range indicator (..) when included in a range. The most significant digit is on the left, the least significant digit is on the right.

10.3.5 DSD bit order

The first sampled DSD bit is stored in the most significant bit of a byte. See subclause 10.6.1.1.

10.3.6 DSD Polarity

A DSD bit equal to one means "plus". A DSD bit equal to zero means "minus".

10.3.7 Hex notation

All Hexadecimal values are preceded by a $. The most significant nibble is on the left, the least significant nibble is on the right.

10.3.8 Range

Constant_1..Constant_2 denotes the range from and including Constant_1 up to and including Constant_2, in increments of 1.

10.3.9 Until

Until is used in figures to indicate that for a structure Byte Positions are used up to (not including) a given value.

At Byte Position B1, the expression “until B2” specifies B2-B1 bytes. At Byte Position B1, the expression “until eos” specifies the number of bytes from B1 up to and including the last byte of the current Sector. Note that Byte Position is specified relative to the start of the current, or a previous, Sector.

10.4 Basic Types

10.4.1 BsMsbf

Bit Sequence, Most Significant Bit First, must be interpreted as a Bit String.

10.4.2 Char

A one-byte character, encoded according to ISO 646. The NUL ($00) character is not allowed for Char.

10.4.3 SiMsbf

Bit sequence, Most Significant Bit First, must be interpreted as Signed Integer using two’s complement notation.

10.4.4 UiMsbf

Bit sequence, Most Significant Bit First, must be interpreted as Unsigned Integer.

10.4.5 Uintn

An n bit, binary encoded, unsigned numerical value.

10.4.6 Uint8

An 8 bit, binary encoded, unsigned numerical value. A Uint8 value must be recorded in a one-byte field.

10.4.7 Uint16

A 16-bit, binary encoded, unsigned numerical value. A Uint16 value, represented by the hexadecimal representation $wxyz, must be recorded in a two-byte field as $wx $yz (most significant byte first).

10.4.8 Uint32

A 32-bit, binary encoded, unsigned numerical value. A Uint32 value, represented by the hexadecimal representation $stuvwxyz, must be recorded in a four-byte field as $st $uv $wx $yz (most significant byte first).

10.5 Payloads for the audio object

Table 10.1 — Syntax of Audio_Frame()

Syntax	Bits	Mnemonics

DSTSpecificConfig(channelConfiguration) {
  if (DSD_Coded)
  {
  DSD()
  }
  if (DST_Coded)
  {
  DST()
  }
}
		
		DSD
		
		DST

Table 10.2 — Syntax of DSD

Syntax	Bits	Mnemonics

DSD() {
  For (Byte_Nr=0; Byte_Nr<Frame_Length; Byte_Nr++)
  {
  For (Channel_Nr=1; Channel_Nr<=N_Channels;
  Channel_Nr++)
  {
  DSD_Byte[Channel_Nr][Byte_Nr]
  }
  }
}
	
	1
		
		Audio_Byte

Table 10.3 — Syntax of DST

Syntax	Bits	Mnemonics
DST() {		
Processing_Mode	**1**	**BsMsbf**
if (Processing_Mode == 0)		
{		
DST_X_Bit	**1**	**BsMsbf**
Reserved	**6**	**BsMsbf**
DSD()		**DSD**
}		
else		
{		
Segmentation()		**Segmentation**
Mapping()		**Mapping**
Half_Probability()		**Half_Probability**
Filter_Coef_Sets()		**Filter_Coef_Sets**
Probability_Tables()		**Probability_Tables**
Arithmetic_Coded_Data()		**Arithmetic_Coded_Data**
}		
}		

Table 10.4 — Syntax of Segmentation

Syntax	Bits	Mnemonics
Segmentation() {		
Same_Segmentation	1	
if(Same_Segmentation == 0)		
{		
Filter_Segmentation()		**Segment_Alloc**
Ptable_Segmentation()		**Segment_Alloc**
}		
else		
{		
Filter_And_Ptable_Segmentation()		**Segment_Alloc**
}		
}		

Table 10.5 — Syntax of Segments

Syntax	Bits	Mnemonics
Segment_Alloc() {		
Resolution_Read = false		
Same_Segm_For_All_Channels	**1**	
if(Same_Segm_For_All_Channels == 0)		
{		
for (Channel_Nr=1; Channel_Nr<=N_Channels; Channel_Nr++)		
{		
Channel_Segmentation()[Channel_Nr]		**Channel_Segmentation**
}		
}		
else		
{		
Channel_Segmentation()[1]		**Channel_Segmentation**
}		
}		

Table 10.6 — Syntax of Channel_Segmentation

Syntax	Bits	Mnemonics
Channel_Segmentation() {		
Nr_Of_Segments = 1		

Start[1] = 0		
End_Of_Channel_Segm	**1**	
while(End_Of_Channel_Segm == 0)		
{		
if (*Resolution_Read == false*)		
{		
Resolution	**13**	**UiMsbf**
Resolution_Read = true		
}		
Scaled_Length[*Nr_Of_Segments*]	**1..13**	**UiMsbf**
*Segment_Length[Nr_Of_Segments] = Resolution * Scaled_Length[Nr_Of_Segments]*		
Start[Nr_Of_Segments+1] = Start[Nr_Of_Segments] + Segment_Length[Nr_Of_Segments]		
Nr_Of_Segments++		
End_Of_Channel_Segm	**1**	
}		
Segment_Length[Nr_Of_Segments] = Frame_Length - Start[Nr_Of_Segments]		
}		

Table 10.7 — Syntax of Mapping

Syntax	Bits	Mnemonics
Mapping() {		
Same_Mapping	**1**	
if(Same_Mapping == 0)		
{		
Filter_Mapping()		**Maps**
Ptable_Mapping()		**Maps**
}		
else		
{		
Filter_And_Ptable_Mapping()		**Maps**
}		
}		

Table 10.8 — Syntax of Maps

Syntax	Bits	Mnemonics
Maps() {		
Nr_Of_Elements = 0		
Same_Maps_For_All_Channels	**1**	
if(Same_Maps_For_All_Channels == 0)		
{		
for (Channel_Nr=1; Channel_Nr<=N_Channels; Channel_Nr++)		
{		
Channel_Mapping()[Channel_Nr]		**Channel_Mapping**
}		
}		
else		
{		
Channel_Mapping()[1]		**Channel_Mapping**
}		
}		

Table 10.9 — Syntax of Channel_Mapping

Syntax	Bits	Mnemonics
Channel_Mapping() {		
for(Seg_Nr=1; Seg_Nr<=Nr_Of_Segments[Channel_Nr]; Seg_Nr++)		

{
Element[Channel_Nr][Seg_Nr]
if (Element[Channel_Nr][Seg_Nr] == Nr_Of_Elements)
{
Nr_Of_Elements++
}
}
}
	0..4	**UiMsbf**

Table 10.10 — Syntax of Half_Probability

Syntax	Bits	Mnemonics

Half_Probability() {
for (Channel_Nr=1; Channel_Nr<=N_Channels; Channel_Nr++)
{
Half_Prob[Channel_Nr]
}
}
	1	**BsMsbf**

Table 10.11 — Syntax of Arithmetic_Coded_Data

Syntax	Bits	Mnemonics

Arithmetic_Coded_Data() {
j=0
do
{
A_Data[j]
j++
} until end of Audio_Frame
}
	1	**BsMsbf**

Table 10.12 — Syntax of Filter_Coef_Sets

Syntax	Bits	Mnemonics

Filter_Coef_Sets() {
for (Filter_Nr=0; Filter_Nr<Nr_Of_Filters; Filter_Nr++)
{		

Coded_Pred_Order	**7**	**UiMsbf**
Pred_Order[Filter_Nr]=Coded_Pred_Order+1		
Coded_Filter_Coef_Set	**1**	**BsMsbf**
if (Coded_Filter_Coef_Set==0)		
{		
for (Coef_Nr=0; Coef_Nr<Pred_Order[Filter_Nr]; Coef_Nr++)		
{		
Coef[Filter_Nr][Coef_Nr]	**9**	**SiMsbf**
}		
}		
else		
{		
CC_Method	**2**	**BsMsbf**
for (Coef_Nr=0; Coef_Nr<CCPO; Coef_Nr++)		
{		
Coef[Filter_Nr][Coef_Nr]	**9**	**SiMsbf**
}		
CCM	**3**	**UiMsbf**
for (Coef_Nr=CCPO; Coef_Nr<Pred_Order[Filter_Nr]; Coef_Nr++)		
{		
Run_Length=0		
do		
{		
RL_Bit	**1**	**BsMsbf**

if (RL_Bit==0)		
{		
Run_Length++		
}		
} while (RL_Bit==0)		
LSBs	**0..6**	**UiMsbf**
Delta=(Run_Length<<CCM)+LSBs		
if (*Delta!=0*)		
{		
Sign	**1**	**BsMsbf**
if (Sign==1)		
{		
Delta = –Delta		
}		
}		
Coef[Filter_Nr][Coef_Nr] = Delta		
Delta8 = 0		
for (Tap_Nr=0; Tap_Nr<CCPO; Tap_Nr++)		
{		
*Delta8 += 8*CCPC[Tap_Nr]*Coef[Filter_Nr][Coef_Nr-Tap_Nr-1]*		
}		
if (Delta8>=0)		
{		
Coef[Filter_Nr][Coef_Nr] -= trunc((Delta8+4)/8)		
}		
else		
{		
Coef[Filter_Nr][Coef_Nr] += trunc((-Delta8+3)/8)		
}		
}		
}		
}		
}		

Table 10.13 — Syntax of Probability_Tables

Syntax	Bits	Mnemonics
Probability_Tables() {		
for (Ptable_Nr=0; Ptable_Nr<Nr_Of_Ptables; Ptable_Nr++)		
{		
Coded_Ptable_Len	**6**	**UiMsbf**
Ptable_Len[Ptable_Nr] = Coded_Ptable_Len+1		
if (Ptable_Len[Ptable_Nr] == 1)		
{		
P_one[Ptable_Nr][0] = 128		
}		
else		
{		
Coded_Ptable	**1**	**BsMsbf**
if (Coded_Ptable==0)		
{		
for (Entry_Nr=0; Entry_Nr<Ptable_Len[Ptable_Nr]; Entry_Nr++)		
{		
Coded_P_one	**7**	**UiMsbf**
P_one[Ptable_Nr][Entry_Nr] = Coded_P_one+1		
}		
}		
else		
{		
PC_Method	**2**	**BsMsbf**

```
                for (Entry_Nr=0; Entry_Nr<PCPO; Entry_Nr++)
                {
                    Coded_P_one                                          7        UiMsbf
                    P_one[Ptable_Nr][Entry_Nr] = Coded_P_one+1
                }
                PCM                                                      3        UiMsbf
                for (Entry_Nr=PCPO;Entry_Nr<Ptable_Len[Ptable_Nr];Entry_Nr++)
                {
                    Run_Length=0
                    do
                    {
                        RL_Bit                                           1        BsMsbf
                        if (RL_Bit==0)
                        {
                          Run_Length++
                        }
                    } while (RL_Bit==0)
                    LSBs                                                 0..4     UiMsbf
                    Delta = (Run_Length<<PCM)+LSBs
                    if (Delta != 0)
                    {
                        Sign                                             1        BsMsbf
                        if (Sign==1)
                        {
                          Delta = – Delta
                        }
                    }
                    P_one[Ptable_Nr][Entry_Nr] = Delta
                    for (Tap_Nr=0; Tap_Nr<PCPO; Tap_Nr++)
                    {
                        P_one[Ptable_Nr][Entry_Nr] –=
                          PCPC[Tap_Nr]*P_one[Ptable_Nr][Entry_Nr-Tap_Nr-1]
                    }
                }
            }
        }
    }
}
```

10.6 Semantics

10.6.1 Audio Streams

An Audio Stream contains the DSD audio signal. Plain DSD or DST coded DSD. An Audio Stream is the concatenation of all Audio Frames in a Byte Stream.

10.6.1.1 DSD Sampled Bit Stream Data

For a 2-Channel Audio signal, the DSD Sampled Bit sequence is defined as follows: L_0, L_1, L_2, L_3, L_4, L_5, L_6, L_7, R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, L_8, L_9, L_{10}, … Where L_0 is the first sampled bit of the left channel of an Audio Frame.

Per Audio Channel, eight successively sampled bits are grouped into one Audio Byte. The most significant bit of an Audio Byte is the first sampled bit of that byte.

10.6.1.2 Structure of the Audio Stream

DST coded Audio Frames have a variable length. The reference model of the DST Decoder is defined in section 10.7.

10.6.1.3 Audio_Frame

Audio_Frame contains the DST or Plain DSD coded audio information for one Frame. The maximum size of an Audio_Frame is equal to the size of a Plain DSD coded Audio_Frame plus one byte. The syntax of an Audio_Frame is defined in Table 10.1.

10.6.1.3.1 DSD

DSD contains the audio data for one Plain DSD Audio_Frame. The syntax of DSD is defined in Table 10.2. An example of a 5 channel DSD Frame is given in Figure 10.2. The definition of Audio_Byte is given in subclause 10.6.1.1.

Channel_Nr is the number of the Audio Channel.

N_Channels is the number of audio channels used as given by the channelConfiguration.

Frame_Length is the length of an Audio Frame in bytes per audio channel. The Frame_Length can be calculated from the Sampling Frequency with the following formula:

$$\text{Frame\_Length} = \frac{\text{Sampling Frequency [Hz]}}{75 * 8} \quad \text{bytes per Audio Channel}$$

The Sampling Frequency can be: 64*44100 Hz, 128*44100 Hz or 256*44100 Hz. The relation between the Frame_Length and Sampling Frequency is provided in Table 10.14.

Table 10.14 — Frame_Length versus Sampling Frequency

Sampling Frequency [Hz]	Frame_Length [bytes]
64*44100	4704
128*44100	9408
256*44100	18816

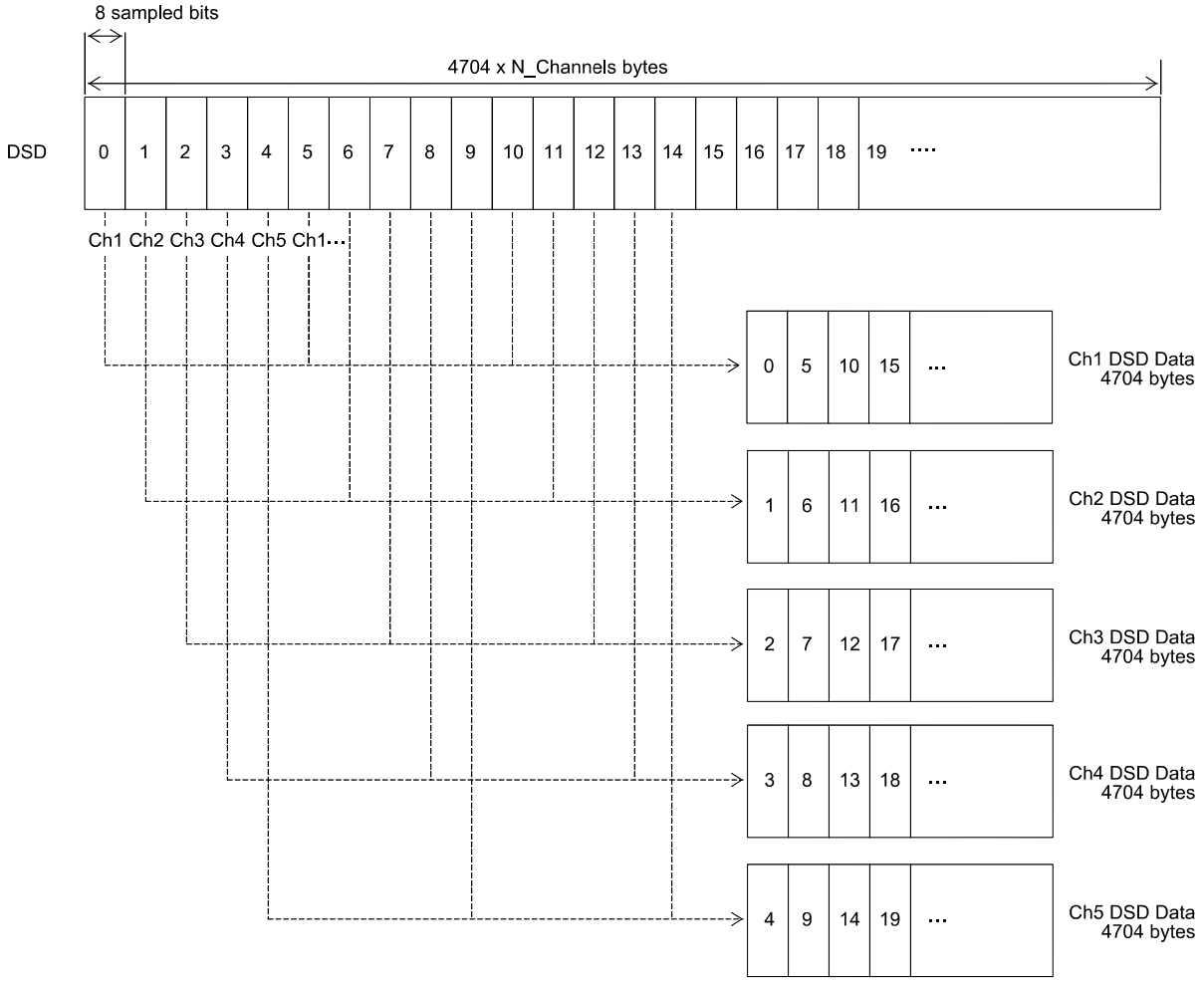


Figure 10.2 — Example of a 5 channel DSD Frame, with Sampling Frequency 64*44100Hz

10.6.1.3.1.1 DSD_Byte

DSD_Byte[Channel_Nr][Byte_Nr] contains the DSD signal as defined in subclause 10.6.1.1.

10.6.1.3.2 DST

DST contains the audio data for one DST coded Audio_Frame. The syntax of DST is defined in Table 10.3.

10.6.1.3.2.1 Processing_Mode

If the Processing_Mode bit is set to one, the Audio_Frame contains the DST_X_Bit and the DSD signal in a lossless coded form. If the Processing_Mode bit is set to zero, the Audio_Frame contains the DST_X_Bit and the DSD signal without lossless coding.

10.6.1.3.2.2 DST_X_Bit

If Frame_Format DST_Coded, each Audio_Frame contains one DST_X_Bit. At encoding the DST_X_Bit shall be set to zero. A reader shall ignore the content of the DST_X_Bit.

10.6.1.3.2.3 Reserved

This value shall be set to zero.

10.6.1.3.2.4 DSD

See subclause 10.6.1.3.1.

10.6.1.3.2.5 Segmentation

For each Audio Channel, the Audio Frame is partitioned into one or more Segments for Filters, and one or more Segments for Ptables. Each Segment may use a different Prediction Filter / Ptable. An example of Segmentation is shown in Figure 10.3. The syntax of Segmentation is defined in Table 10.4.

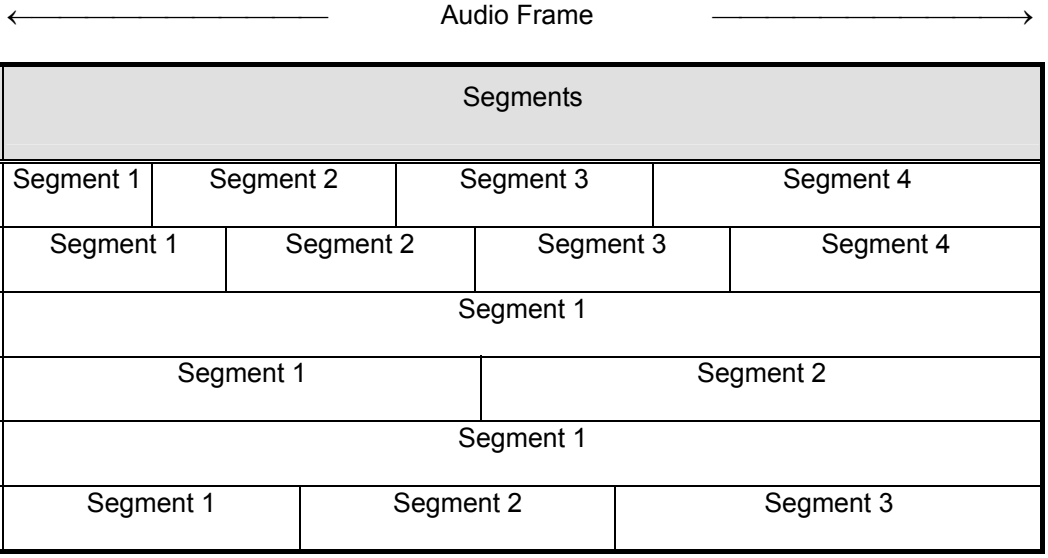


Figure 10.3 — Example of Segmentation

Filter_Segmentation

For each Audio Channel, the Audio Frame is partitioned into one or more Segments for Prediction Filters. Each Segment may use a different Prediction Filter. The variables Nr_Of_Segments[ ] and Segment_Length[ ][ ] from Segment_Alloc (see subclause 10.6.1.3.2.5.2) used for Filter_Segmentation are referred to as:

- Filters.Nr_Of_Segments[Channel_Nr] and
- Filters.Segment_Length[Channel_Nr][1..Filters.Nr_Of_Segments[Channel_Nr]].

With Channel_Nr = 1..N_Channels.

Ptable_Segmentation

For each Audio Channel, the Audio Frame is partitioned into one or more Segments for Ptables. Each Segment may use a different Ptable. The variables Nr_Of_Segments[ ] and Segment_Length[ ][ ] from Segment_Alloc (see subclause 10.6.1.3.2.5.2) used for Ptable_Segmentation are referred to as:

- Ptables.Nr_Of_Segments[Channel_Nr] and
- Ptables.Segment_Length[Channel_Nr][1..Ptables.Nr_Of_Segments[Channel_Nr]].

With Channel_Nr = 1..N_Channels.

Filter_And_Ptable_Segmentation

For each Audio Channel, the Audio Frame is partitioned into one or more Segments. Each Segment may use a different combination of Prediction Filter and Ptable. For each Audio Channel, the following equations must be true:

Filters.Nr_Of_Segments[Channel_Nr] = Ptables.Nr_Of_Segments[Channel_Nr] = Nr_Of_Segments[Channel_Nr]

Filters.Segment_Length[Channel_Nr][] = Ptables.Segment_Length[Channel_Nr][] = Segment_Length[Channel_Nr][]

With Channel_Nr = 1..N_Channels.

10.6.1.3.2.5.1 Same_Segmentation

If Same_Segmentation is one, the Ptables and Prediction Filters use one and the same Segmentation. If Same_Segmentation is zero, the partitioning for Prediction Filters is independent from the partitioning for Ptables, for the Audio Frame.

10.6.1.3.2.5.2 Segment_Alloc

Segment_Alloc defines the Segmentation for the Prediction Filters and/or the Ptables. The syntax of Segment_Alloc is defined in Table 10.5.

For each Audio Channel, the variables Nr_Of_Segments and Segment_Length[1..Nr_Of_Segments] from Channel_Segmentation (see subclause 10.6.1.3.2.5.2.2) are referred to as:

- Nr_Of_Segments[Channel_Nr]
- Segment_Length[Channel_Nr][1..Nr_Of_Segments[Channel_Nr]]

In the syntax diagrams, syntax variables are shown in *italics*.

Resolution_Read indicates whether Resolution from Channel_Segmentation, see subclause 10.6.1.3.2.5.2.2, has been read. Resolution_Read is set to true in the Channel_Segmentation of the first Audio Channel with more than one Segment. Note that if Prediction Filters and Ptables use independent Segmentation, they also use an independent Resolution_Read.

Channel_Nr is a local index variable.

N_Channels is the number of audio channels used.

10.6.1.3.2.5.2.1 Same_Segm_For_All_Channels

If Same_Segm_For_All_Channels is one, only the Segmentation for the first Audio Channel is stored and Channel_Segmentation()[Channel_Nr] = Channel_Segmentation()[1] for all Audio Channels. If Same_Segm_For_All_Channels is zero, the Audio Frame is partitioned into segments independent for each Audio Channel.

10.6.1.3.2.5.2.2 Channel_Segmentation

Channel_Segmentation defines the Segmentation of the Prediction Filters and/or the Ptables. The syntax of Channel_Segmentation is defined in Table 10.6.

The following variables are used in the syntax of Channel_Segmentation:

- Nr_Of_Segments
- Start[1..Nr_Of_Segments]

- Segment_Length[1..Nr_Of_Segments]

Nr_Of_Segments is the number of Segments for the current Audio Channel. The maximum number of Segments is MAXNRSEGS. MAXNRSEGS must be 4 for the Filter_Segmentation, 8 for the Ptable_Segmentation, and 4 for the Filter_And_Ptable_Segmentation.

Resolution_Read indicates whether the variable Resolution has been read in this or a previous Channel_Segmentation. Resolution_Read is set to true in the Channel_Segmentation of the first Audio Channel with more than one Segment. Note that if Prediction Filters and Ptables use independent Segmentation, they also use an independent Resolution_Read.

Segment_Length[Seg_Nr] contains the length of the Segment in bytes, where:

1 <= Seg_Nr <= Nr_Of_Segments.

Start[Seg_Nr] is the starting position in bytes of Segment[Seg_Nr].

Frame_Length, see subclause 10.6.1.3.1.

10.6.1.3.2.5.2.2.1 End_Of_Channel_Segm

If End_Of_Channel_Segm is zero, one or more values for Scaled_Length will follow. If End_Of_Channel_Segm is one, the Channel_Segmentation structure ends.

10.6.1.3.2.5.2.2.2 Resolution

Each value of Scaled_Length is multiplied by Resolution to get the Segment length in bytes. Resolution is stored only once, at the beginning of the first Audio Channel with more than one Segment. If all Audio Channels have only one Segment, Resolution is not encoded.

Resolution must be in the range of 1 to Frame_Length - MINSEGLEN. MINSEGLEN must be 128 bytes for Filter_Segmentation, 4 bytes for Ptable_Segmentation, and 128 bytes for Filter_And_Ptable_Segmentation.

10.6.1.3.2.5.2.2.3 Scaled_Length

For each Segment, except the last one, a value of Scaled_Length is encoded. The length in bytes of a Segment is calculated with the following formula:

Segment_Length[Seg_Nr] = Resolution * Scaled_Length[Seg_Nr],

where,

1 <= S_Nr < Nr_Of_Segments.

The minimum Segment length of each Segment is MINSEGLEN, see subclause 10.6.1.3.2.5.2.2.2.

For Ptable_Segmentation the length of the first Segment of each Audio Channel must be at least (Pred_Order[Filter[Channel_Nr][1]]+7)/8 bytes. For the definition of Filter[ ][ ] see subclause 10.6.1.3.2.6. For the definition of Pred_Order[ ] see subclause 10.6.1.3.2.8.

The number of bits needed to encode Scaled_Length[Seg_Nr] depends on the value of Range. Range must be calculated with the following formula:

$$\text{Range} = \text{Trunc}\left(\frac{\text{Frame\_Length} - \text{Start}[\text{Seg\_Nr}] - \text{MINSEGLEN}}{\text{Resolution}}\right).$$

If $2^{n-1} \leq \text{Range} < 2^n$, n bits must be used to encode Scaled_Length[Seg_Nr], see Table 10.15. The minimum value of Range is 1. The length of the last Segment is not encoded on the disc. The length of the last Segment can be calculated from the Frame Length and the start position of the last Segment with the following formula:

Segment_Length[Nr_Of_Segments] = Frame_Length - Start[Nr_Of_Segments].

Table 10.15 — Bits used to encode Scaled_Length

Range	bits used
1	1
2..3	2
4..7	3
8..15	4
16..31	5
32..63	6
64..127	7

Range	bits used
128..255	8
256..511	9
512..1023	10
1024..2047	11
2048..4095	12
4096..8191	13

10.6.1.3.2.6 Mapping

Mapping defines the Prediction Filters and Ptables used with the Segments specified in subclause 10.6.1.3.2.5. An example of Prediction Filter allocation for the Segmentation example in Figure 10.3 is shown in Figure 10.4. The syntax of Mapping is defined in Table 10.7.

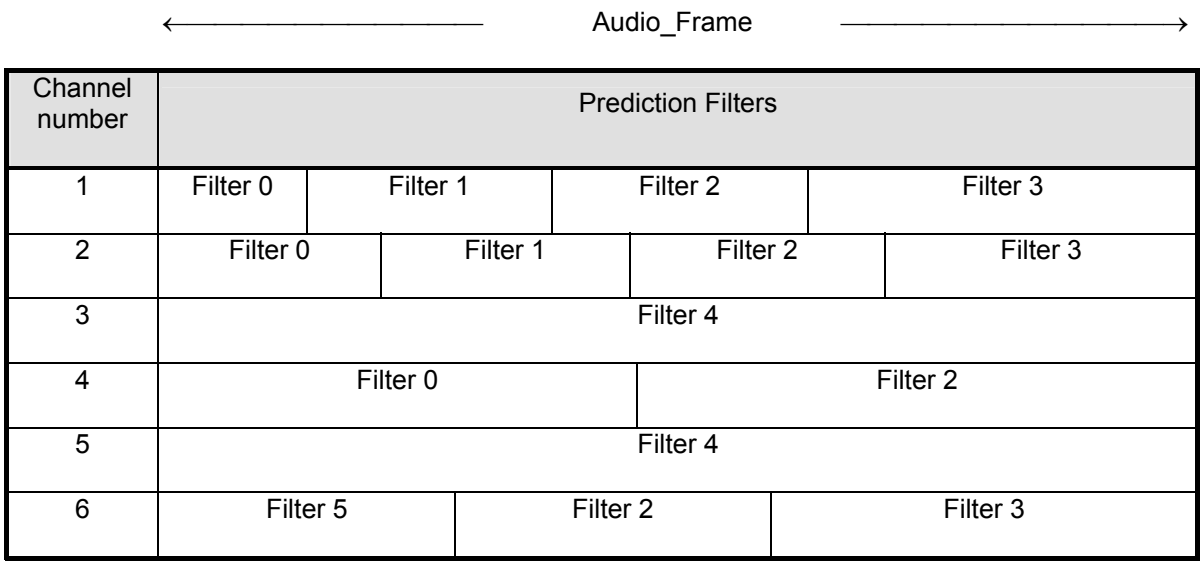


Figure 10.4 — Example of filter allocation

Filter_Mapping

For each Audio Channel and each Segment, a Prediction Filter number is encoded. For Filter_Mapping, the variable Element[ ][ ] from Channel_Mapping (see subclause 10.6.1.3.2.6.2.2) contains the Prediction Filter numbers. For Filter_Mapping, Element[ ][ ] is referred to as:

Filter[Channel_Nr][1..Filters.Nr_Of_Segments[Channel_Nr]].

For Filter_Mapping, the variable Nr_Of_Elements (see subclause 10.6.1.3.2.6.2.2) is referred to as Nr_Of_Filters.

Ptable_Mapping

For each Audio Channel and each Segment, a Ptable number is encoded. For Ptable_Mapping, the variable Element[ ][ ] from Channel_Mapping (see subclause 10.6.1.3.2.6.2.2) contains the Ptable numbers. For Ptable_Mapping, Element[ ][ ] is referred to as:

Ptable[Channel_Nr][1..Ptables.Nr_Of_Segments[Channel_Nr]].

For Ptable_Mapping, the variable Nr_Of_Elements (see subclause 10.6.1.3.2.6.2.2) is referred to as Nr_Of_Ptables.

Filter_And_Ptable_Mapping

For each Audio Channel and each Segment, a common Prediction Filter and Ptable number is encoded. For Filter_and_Ptable_Mapping, the variable Element[ ][ ] from Channel_Mapping (see subclause 10.6.1.3.2.6.2.2) contains the common Prediction Filter and Ptable numbers. For Filter_and_Ptable_Mapping, Element[ ][ ] is referred to as:

Filter[Channel_Nr][1..Filters.Nr_Of_Segments[Channel_Nr]],

as well as:

Ptable[Channel_Nr][1..Ptables.Nr_Of_Segments[Channel_Nr]].

For Filter_and_Ptable_Mapping, the variable Nr_Of_Elements (see subclause 10.6.1.3.2.6.2.2) is referred to as Nr_Of_Filters as well as Nr_Of_Ptables.

10.6.1.3.2.6.1 Same_Mapping

If Same_Mapping is one, the Ptables and Prediction Filters use one and the same Mapping. If Same_Mapping is zero, the mapping for Prediction Filters is independent from the mapping for Ptables, for the Audio Frame.

10.6.1.3.2.6.2 Maps

Maps defines the mapping of Prediction Filters and Ptables to the Segments defined in subclause 10.6.1.3.2.5. The syntax of Maps is defined in Table 10.8.

The following variables are used in the syntax of Maps:

- For each Audio Channel, Element[Channel_Nr][1..Nr_Of_Segments[Channel_Nr]]
- Nr_Of_Elements

Nr_Of_Elements is the total number of Prediction Filter and/or Ptables, used in Maps. Nr_Of_Elements must be in the range 1 .. 2 * N_Channels.

Channel_Nr is a local index variable, used in Table 10.8 and Table 10.9.

N_Channels is the number of audio channels used.

10.6.1.3.2.6.2.1 Same_Maps_For_All_Channels

If Same_Maps_For_All_Channels is one, only Element[1][ ] is stored and each Audio Channel uses the same array Element[Channel_Nr][ ] = Element[1][ ]. If Same_Maps_For_All_Channels is equal to zero,

Element[Channel_Nr][ ] is stored independent for each Audio Channel. If Nr_Of_Segments[Channel_Nr] does not have the same value for all Audio Channels, Same_Maps_For_All_Channels must be zero.

10.6.1.3.2.6.2.2 Channel_Mapping

Channel_Mapping contains per Audio Channel the Prediction Filter and/or Ptable numbers used for each Segment. The syntax of Channel_Mapping is defined in Table 10.9.

Nr_Of_Elements is the total number of Prediction Filters and/or Ptables for all channels. Nr_Of_Elements is initialized in Maps, see Table 10.8.

Channel_Nr is the index variable from Maps, see Table 10.8.

Seg_Nr is a local index variable.

Nr_Of_Segments[Channel_Nr] is the total number of Segments used in the current Audio Frame for Audio Channel Channel_Nr.

10.6.1.3.2.6.2.2.1 Element

Element is the Prediction Filter and/or Ptable number used in the Segment. The number of bits used to encode Element depends on the value of Nr_Of_Elements. In each iteration, Element must be <= Nr_Of_Elements. Therefore, Element[1][1] is always zero and is not stored (#bits = 0). For all other Audio Channels and Segments, Nr_Of_Elements > 0 and the number of bits needed to store Element is n with: $2^{n-1} \le Nr\_Of\_Elements < 2^n$, see Table 10.16.

Table 10.16 — Bits used to encode Element

Nr_Of_Elements	#bits used
0	0
1	1
2..3	2
4..7	3
8..12	4

10.6.1.3.2.7 Half_Probability

The syntax of Half_Probability is defined in Table 10.10.

Channel_Nr is a local index variable.

N_Channels is the number of audio channels used.

10.6.1.3.2.7.1 Half_Prob

Half_Prob is used to encode, for each Audio Channel, which method will be used for applying a probability value to the arithmetic decoder. The definition of Half_Prob is given in Table 10.17.

Table 10.17 — Definition of Half_Prob

Half_Prob[ ]	Probability to use during first Pred_Order[ ] bits of the Audio Channel
0	Use the entries from the Ptable.
1	Use p=½ (corresponds to P_one = 128).

For optimum coding gain it is desired that the next residual bit in E has the value that has the greatest probability. If a probability is applied that reflects a high chance of the next E bit being 1 while the next E bit is a 0, then more than 1 bit is required in the arithmetic code to send this bit.

The prediction filter is initially filled with an initialization pattern. During the first Pred_Order[Filter[Channel_Nr][1]] samples in Audio Channel Channel_Nr, the Prediction Filter is gradually filled with real DSD data. As a consequence the probability distribution can be quite different from the rest of the frame, and the combination of applied E and P for these bits results in more bits than desired. When applying a probability of ½ during encoding, each bit will also cost only one bit in the arithmetic code.

Therefore Half_Prob is available for each channel separately to be able to overrule a bad combination of E and P at the start of a frame.

10.6.1.3.2.8 Filter_Coef_Sets

For each Segment in each Audio Channel, the DST decoder uses a Prediction Filter. In case two or more Prediction Filters are equal, the corresponding filter coefficients may be encoded only once. The variables used in the syntax of Filter_Coef_Sets are:

- Pred_Order[Filter_Nr]
- Coef[Filter_Nr][0..Pred_Order[Filter_Nr]-1]

With Filter_Nr = 0..Nr_Of_Filters-1.

All Prediction Filter coefficients are encoded in the disc. Per Prediction Filter, the order of prediction (the number of coefficients) and the coefficients are encoded. The Prediction Filter coefficients can be coded using simple linear prediction and Rice coding. Rice coding is a variable length coding technique (special case of Huffman coding) which is used to decrease the number of bits required for a certain “message”, without loss of information. The syntax of Filter_Coef_Sets is defined in Table 10.12.

The least significant bit of Coef[0][0] is called DST_Y_Bit, see subclause 10.6.1.3.2.8.1.

Nr_Of_Filters is the value calculated in Mapping, see subclause 10.6.1.3.2.6.

Pred_Order[ ] is an array that contains the prediction order for each Prediction Filter. Where Pred_Order[Filter_Nr] = Coded_Pred_Order + 1, for Filter_Nr$\in$(0..Nr_Of_Filters-1). The allowed range of the prediction order is: $1 \leq$ Pred_Order[Filter_Nr] ≤ 128.

Coef[ ][ ] is a two-dimensional array that contains all coefficients of all Prediction Filters. Each entry of Coef[ ][ ] must be in the range of -256 to +255. The first (left) index is the Filter_Nr and ranges from 0 through Nr_Of_Filters-1. The second (right) index is the coefficient number and ranges from 0 through Pred_Order[Filter_Nr]-1.

CCPO is the Coefficient Coding Prediction Order (CCPO). The relation between CC_Method and CCPO is defined in Table 10.18.

Table 10.18 — Relation between CC_Method and CCPO

CC_Method	CCPO
'00'	1
'01'	2
'10'	3
'11'	Not used

The restriction CCPO < Pred_Order[Filter_Nr] applies.

Run_Length is a help variable to count the number of zeros in the run length code that is part of the Rice code.

Delta is a help variable to calculate the Rice coded Number.

Delta8 is a help variable to calculate the Rice coded Number.

CCPC[] is an array that contains the Coefficient Coding Prediction Coefficients (CCPC) that are used for the linear prediction of the Filter coefficients. The relation between CC_Method and CCPC[] is defined in Table 10.19.

Table 10.19 — Relation between CC_Method and CCPC[]

CC_Method	CCPC[0]	CCPC[1]	CCPC[2]
'00'	-1	-	-
'01'	-2	1	-
'10'	-9/8	-5/8	6/8
'11'	Not used	Not used	Not used

The linear prediction requires rounding, as specified in the syntax of Table 10.12.

10.6.1.3.2.8.1 DST_Y_Bit

DST_Y_Bit is the least significant bit of Coef[0][0]. At encoding the DST_Y_Bit shall be set to one. A reader shall ignore the content of the DST_Y_Bit.

10.6.1.3.2.8.2 Coded_Pred_Order

Coded_Pred_Order is a 7 bit unsigned integer that contains the coded prediction order of the current Prediction Filter.

10.6.1.3.2.8.3 Coded_Filter_Coef_Set

Coded_Filter_Coef_Set indicates whether the Prediction Filter coefficients are predicted and Rice coded. Coded_Filter_Coef_Set is set to zero if the Prediction Filter coefficients are stored directly.

Coded_Filter_Coef_Set is set to one if the Prediction Filter coefficients are predicted and Rice coded.

The maximum number of bits permitted for a single Prediction Filter inside Filter_Coef_Sets is:

7+1+Pred_Order[ ]*9,

where 7 counts the bits for Coded_Pred_Order, 1 counts the Coded_Filter_Coef_Set bit and the rest the Pred_Order[] coefficients of 9 bit each.

10.6.1.3.2.8.4 CC_Method

CC_Method is a 2 bit code that identifies the Coefficient Coding Method of the current Prediction Filter.

10.6.1.3.2.8.5 CCM

CCM is a 3 bit unsigned integer that contains the Coefficient Coding M parameter that is used for Rice decoding the coefficients of the current Prediction Filter. The minimum allowed value for CCM is zero. The maximum allowed value for CCM is 6.

10.6.1.3.2.8.6 RL_Bit

RL_Bit is used to retrieve the single bits of the run length code that consists of zeros with a terminating one. The shortest run length code is '1'.

10.6.1.3.2.8.7 LSBs

CCM least significant bits of the absolute value of the predicted coefficient are read from the stream directly and are stored in LSBs.

10.6.1.3.2.8.8 Sign

Sign is a bit that indicates if the predicted coefficient is positive (Sign='0') or negative (Sign='1').

10.6.1.3.2.9 Probability_Tables

For each Segment in each Audio Channel, the decoder uses a Probability Table (Ptable). In case two or more probability tables are equal, the corresponding probability table entries may be available from the stream only once. The variables used in the syntax of Probability_Tables are:

- Ptable_Len[Ptable_Nr]
- P_One[Ptable_Nr][0…Ptable_Len[Ptable_Nr]-1]

With Ptable_Nr = 0..Nr_Of_Ptables-1.

In Probability_Tables all probability table entries are encoded. Per probability table, the length of the table (= the number of entries) and the entries are encoded. The Ptable entries can be coded using simple linear prediction and Rice coding. The syntax of Probability_Tables is defined in Table 10.13.

Nr_Of_Ptables is the value calculated in Mapping, see subclause 10.6.1.3.2.6.

Ptable_Len[ ] is an array that contains the probability table length for each Ptable. Where Ptable_Len[Ptable_Nr] = Coded_Ptable_Len + 1, for Ptable_Nr$\in${0..Nr_Of_Ptables-1}. The allowed range of Ptable length: $1 \leq$ Ptable_Len[Ptable_Nr] ≤ 64.

P_one[ ][ ] is a two-dimensional array that finally contains all entries of all probability tables. The first (left) index is the Ptable_Nr and ranges from 0 through Nr_Of_Ptables-1. The second (right) index is the entry number and ranges from 0 through Ptable_Len[Ptable_Nr]-1. Each entry of P_one[ ][ ] is in the range of 1 to

128, corresponding to a probability of 1/256 to 128/256 of the next error bit (bit E, See Figure 10.5) being a one.

PCPO is the Ptable Coding Prediction Order (PCPO). The relation between PC_Method and PCPO is defined in Table 10.20.

Table 10.20 — Relation between PC_Method and PCPO

PC_Method	PCPO
'00'	1
'01'	2
'10'	3
'11'	Not used

The restriction PCPO < Ptable_Len[Ptable_Nr] applies.

Run_Length is a help variable to count the number of zeros in the run length code that is part of the Rice code.

Delta is a help variable to calculate the Rice decoded Number.

PCPC[] is an array that contains the Ptable Coding Prediction Coefficients (PCPC) that are used for the linear prediction of the Ptable entries. The relation between PC_Method and PCPC[] is defined in Table 10.21.

Table 10.21 — Relation between PC_Method and PCPC[]

PC_Method	PCPC[0]	PCPC[1]	PCPC[2]
'00'	-1	-	-
'01'	-2	1	-
'10'	-3	3	-1
'11'	Not used	Not used	Not used

10.6.1.3.2.9.1 Coded_Ptable_Len

Coded_Ptable_Len is a 6 bit unsigned integer that contains the coded probability table length.

10.6.1.3.2.9.2 Coded_Ptable

Coded_Ptable indicates whether the Ptable entries are predicted and Rice coded. Coded_Ptable is set to zero if the Ptable entries are stored directly. Coded_Ptable is set to one if the Ptable entries are predicted and Rice coded.

The maximum number of bits permitted for a single Ptable inside Probability_Tables is:

6+1+Ptable_Len[ ]*7,

where 6 counts the bits for Coded_Ptable_Len, 1 counts the Coded_Ptable bit and the rest the Ptable_Len[] coded Ptable entries of 7 bit each.

10.6.1.3.2.9.3 Coded_P_one

Coded_P_one is a 7 bit unsigned integer that contains the coded value of the next entry of the current Ptable.

10.6.1.3.2.9.4 PC_Method

PC_Method is a 2 bit field that identifies the Ptable Coding Method of the current Ptable.

10.6.1.3.2.9.5 PCM

PCM is a 3 bit unsigned integer that contains the Ptable Coding M parameter that is used for Rice decoding of the Ptable entries of the current Ptable. The minimum allowed value for PCM is zero. The maximum allowed value for PCM is 4.

10.6.1.3.2.9.6 RL_Bit

RL_Bit contains the single bits of the run length code, that consists of zeros with a terminating one. The shortest run length code is '1'.

10.6.1.3.2.9.7 LSBs

PCM least significant bits of the absolute value of the predicted entry are stored in LSBs.

10.6.1.3.2.9.8 Sign

Sign is a bit that indicates if the predicted entry is positive (Sign='0') or negative (Sign='1').

10.6.1.3.2.10 Arithmetic_Coded_Data

The syntax of Arithmetic_Coded_Data is defined in Table 10.11. Note that the length of Arithmetic_Coded_Data is not encoded.

10.6.1.3.2.10.1 A_Data

A_Data[] contains two parts:

- the Arithmetic Code
- Stuffing Bits

The Stuffing Bits are appended at the end of the Arithmetic Code to align the Audio_Frame to a byte boundary. The number of Stuffing Bits is 0..7. The value of the Stuffing Bits must be zero.

A_Data[] is used by the function "Input next bit D" as described in Annex C. The minimum length of A_Data is zero bits. If the length of A_Data is not equal to zero, A_Data[0] must have the value zero. The maximum length of Arithmetic Code is the number of bits processed by "Input next bit D". It is allowed that trailing zeros of Arithmetic Code are not encoded in A_Data[].

10.7 DST Decoder Reference Model (Normative)

10.7.1 DST Decoder Block Diagrams

In Figure 10.5, the block diagram of a DST Decoder for a mono DSD signal is given. Figure 10.6 is a short hand notation for Figure 10.5. Figure 10.7 shows the diagram that is applicable when C-channels plus the DST_X_Bit have to be decoded.

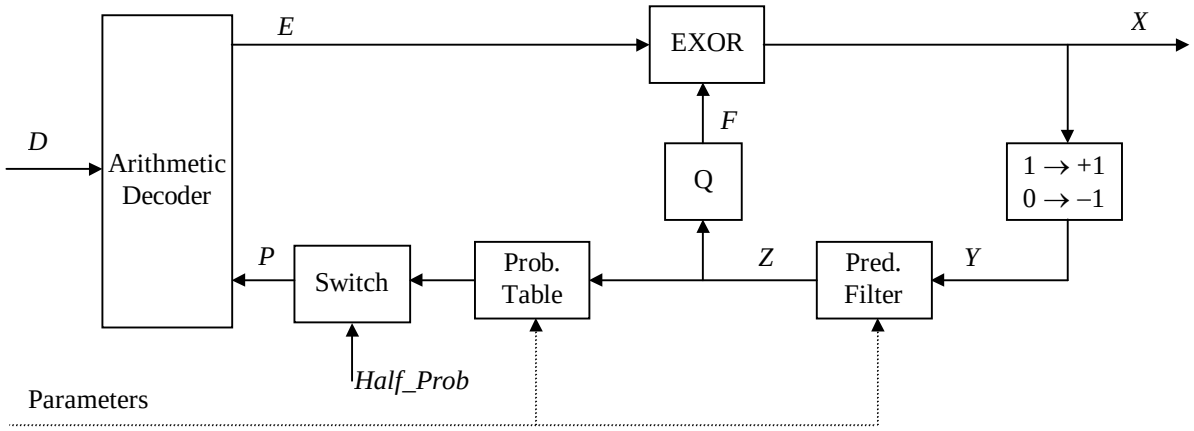


Figure 10.5 — Block diagram of the DST decoder for a mono DSD signal

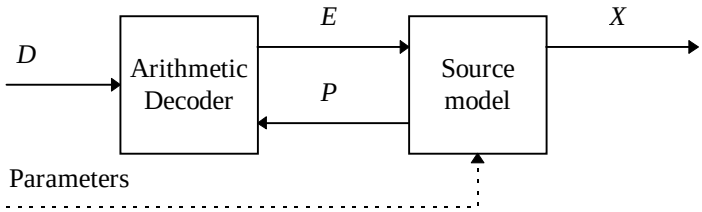


Figure 10.6 — Global diagram of the single channel DST decoder

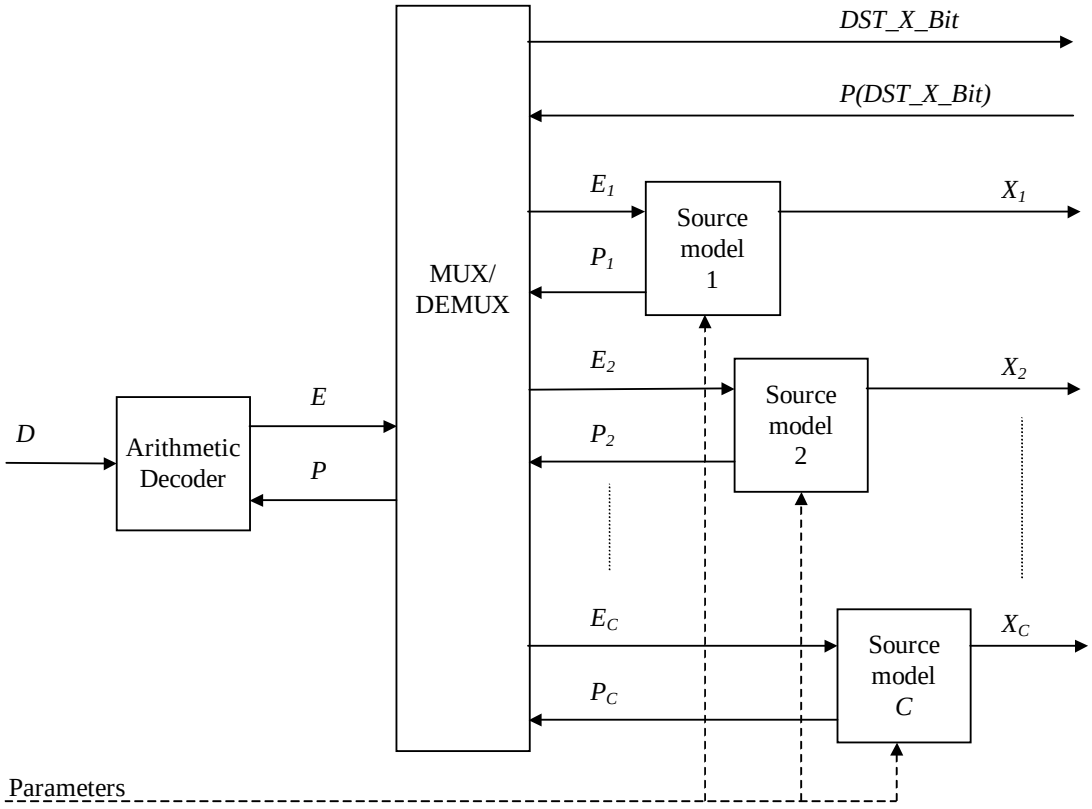


Figure 10.7 — Global diagram of the C-channel DST decoder

10.7.2 DST Decoding Processes

The parameters processed and extracted from the stream are used to decode the DST coded frame. This subclause explains the decoding processes needed for DST coded frames.

10.7.2.1 Introduction

In Figure 10.7, three functions are distinguished: the arithmetic decoder, the multiplexer/demultiplexer and a number of source models. The arithmetic decoder receives the sequence of bits (*D*=A_Data) and the sequence of probabilities (*P*) and generates the sequence of bits (*E*). The *E* and *P* sequences are assigned to the source models in a cyclic order that is controlled by the multiplexer/demultiplexer. Every source model receives the required parameters like prediction filter coefficients and probability table entries from the stream as specified in syntax and semantics.

Source model S corresponds with channel S. The output X of Source model S is the DSD signal for channel S.

10.7.2.2 Arithmetic Decoder

Arithmetic coding is a variable length coding technique to compress data near to its entropy. The coded data is represented as a number. The number uses as many digits as are required to uniquely identify the original data. For more information about arithmetic coding see:

I.H. Witten, R.M. Neal, and J.G. Cleary, “Arithmetic coding for data compression”, Communications ACM, vol. 30, pp. 520-540, June 1987.

P.G. Howard, and J.S. Vitter, “Arithmetic coding for data compression”, Proc. IEEE, vol. 82, no. 6, p. 857-65, June 1994.

Table 10.22 defines the variables used in Figure 10.8, Figure 10.9, Figure 10.10 and Figure 10.11.

Table 10.22 — Variables used in Figure 10.8, Figure 10.9, Figure 10.10 and Figure 10.11

Name	Characteristics	Description
A	12 bit register	This unsigned integer represents the current interval size of the arithmetic decoder.
C	12 bit register	This unsigned integer holds part of the arithmetic code bits.
K	4 bit variable	This unsigned integer is a 4 bit approximation of A.
P	8 bit variable	This unsigned integer is the probability value applied to the arithmetic decoder.
Q	12 bit variable	This unsigned integer is the multiplication of K and P.

Figure 10.8 shows the overall flow chart of the DST decoding algorithm.

The initialization process is shown in Figure 10.9 and is required at the start of decoding each frame. It consists of loading the first 13 bits of the arithmetic code into register C and resetting register A to 4095. The first bit read into C will be overwritten, this is intended because the first bit is always 0.

The function “Input next bit D” in Figure 10.8, Figure 10.9 and Figure 10.11 means bit D is taken from A_Data[ ], starting with the first bit. After all bits from A_Data[ ] have been read, the function “Input next bit D” sets bit D to 0.

The reading of probability information is just retrieving a *P* from the source model (see subclause 10.7.2.3), except for the first bit, where a DST_X_Bit probability is read instead.

Figure 10.10 illustrates the decoding of one bit of *E* and updating of register A and C. First the current approximated interval size, *K*, is calculated. Then the multiplication of the approximated interval size *K* and the applied probability value *P* is stored in *Q*. If C is greater than or equal to A-Q, then the arithmetic code lies in the upper part of the interval which means that the original bit encoded, *E*, was a ‘1’ bit; otherwise a ‘0’ bit was transmitted. A and C must be adjusted in the same way as in the encoder.

The renormalizing process is shown in Figure 10.11. Renormalizing is required when the value of A is too small. If A is too small, both A and C are shifted one bit left, and a new bit of the arithmetic code, *D*, is read into the least significant bit of C.

It is possible that the last bits of A_Data[ ] are not used by the function “Input next bit D” for decoding of the Audio Frame. These unused bits are stuffing bits for alignment of the Audio Frame on a byte boundary.

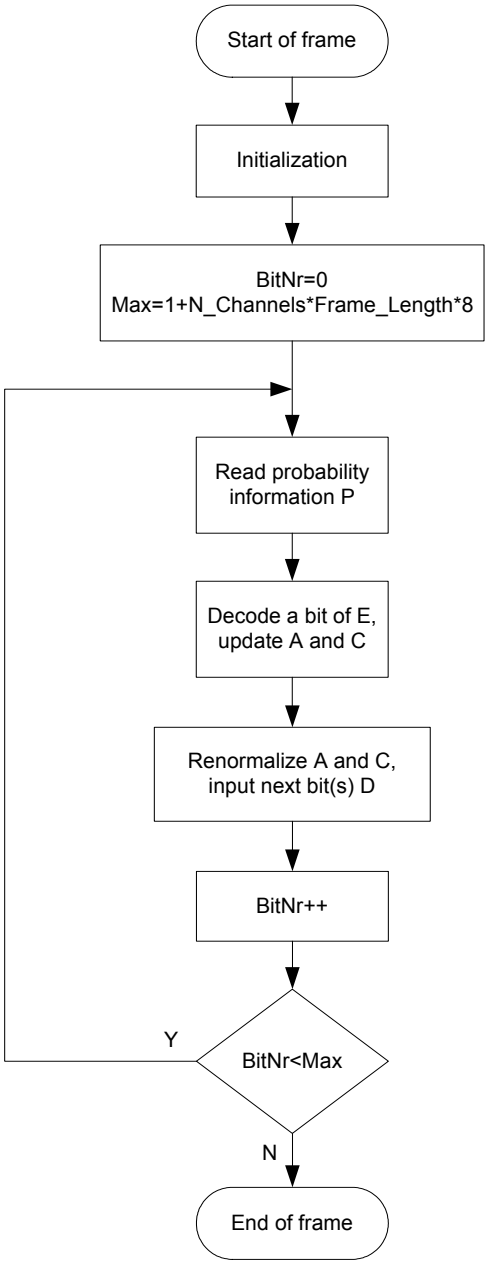


Figure 10.8 — Flowchart of the arithmetic decoder

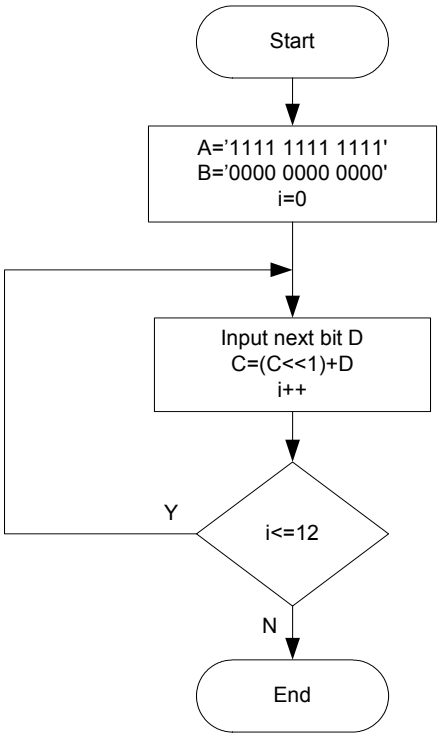


Figure 10.9 — Initialization

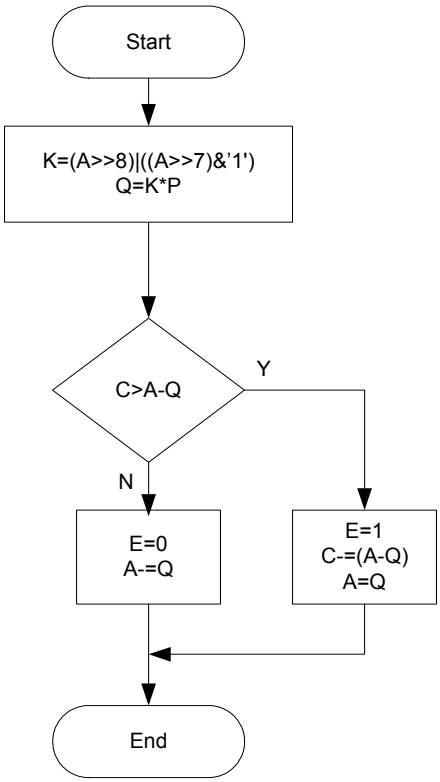


Figure 10.10 — Decode a bit of *E*, update A and C

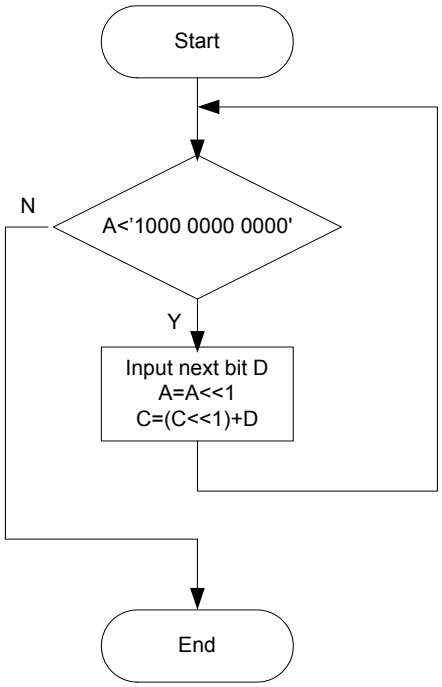


Figure 10.11 — Renormalize A and C, input next bit(s) D

10.7.2.3 Source Model

The decoding process in the source model is described for one channel only, as it is equivalent for all channels. Figure 10.12 shows the block diagram of the source model.

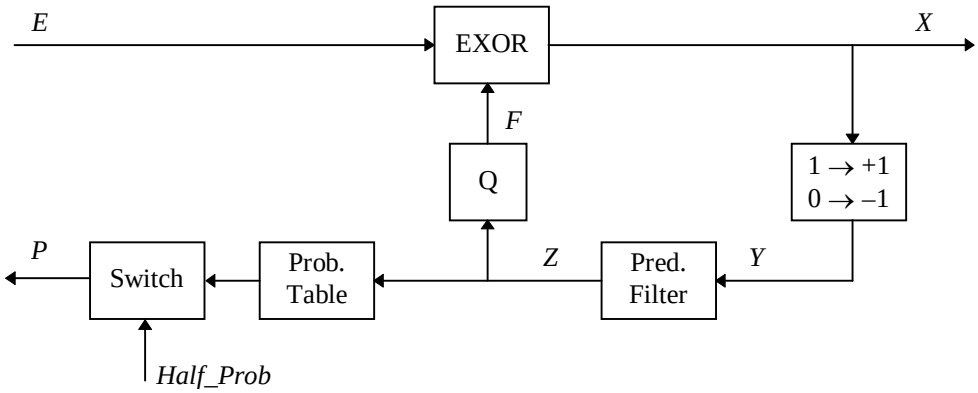


Figure 10.12 — Source model

Segmentation and Mapping determine which Prediction Filter and which Ptable must be used to decode the next bit of an Audio Channel. The two functions Filter_N(n) and Ptable_N(n) return the Prediction Filter and Ptable number used for decoding bit n of an Audio Channel. The functions Filter_N(n) and Ptable_N(n) use Segmentation and Mapping information. Filter_N(n) is defined as follows:

Filters.Start[1] = 0

Filters.Start[Seg+1] = Filters.Start[Seg] + Filters.Segment_Length[Channel_Nr][Seg]

Where: Seg = 1 .. Filters.Nr_Of_Segments[Channel_Nr]

and Seg is the Segment number of Audio Channel Channel_Nr.

For bit n, the variable Seg can be determined via:

Filters.Start[Seg] <= (n >> 3) < Filters.Start[Seg+1]

Filter_N(n) can be found by using the following formula:

Filter_N(n) = Filter[Channel_Nr][Seg]

The function Ptable_N(n) is defined using the same mechanism by:

- Replacing Filters by Ptables
- Replacing Filter_N by Ptable_N

Filters.Segment_Length[ ][ ] and Ptables.Segment_Length[ ][ ] are defined in subclause 10.6.1.3.2.5.

Filters.Nr_Of_Segments[ ] and Ptables.Nr_Of_Segments[ ] are defined in subclause 10.6.1.3.2.5.

Filter[ ][ ] and Ptable[ ][ ] are defined in subclause 10.6.1.3.2.6.

Filter_N(n) and Ptable_N(n) are used in the following definitions, which are used in the next section:

N	Pred_Order[Filter_N(n)]	Prediction order of the Prediction Filter that the current Segment uses.
H [ ]	Coef [Filter_N(n)] [ ]	Coefficients of the Prediction Filter that the current Segment uses.
L	Ptable_Len[Ptable_N(n)]	Length of the Ptable that the current Segment uses.
T [ ]	P_one [Ptable_N(n)] [ ]	Entries of the Ptable that the current Segment uses.
n	bit sequence number, range: 0 .. 8*Frame_Length - 1	Variable that runs through all bits of the current Audio Channel.

Frame_Length is defined in subclause 10.6.1.3.1.

Pred_Order[ ] and Coef[ ][ ] are defined in subclause 10.6.1.3.2.8.

Ptable_Len[ ] and P_One[ ][ ] are defined in subclause 10.6.1.3.2.9.

10.7.2.3.1 Initialization

The initialization of the prediction filter at the start of each frame is defined as:

$$Y[m] = (-1)^m \text{ for } -N \le m < 0$$

The output value of the prediction filter is defined as:

$$Z[n] = \sum_{i=0}^{N-1} Y[n-1-i] \cdot H[i]$$

The Q-function transforms Z into F as follows:

$$F[n] = \begin{cases} 1 & : \quad Z[n] \geq 0 \\ 0 & : \quad Z[n] < 0 \end{cases}$$

There are two methods for applying probability values to the arithmetic decoder. In case *Half_Prob* [*Channel_Nr*] = '0' the probability value is determined by:

$$P[n] = T\left[\min\left(|Z[n]| >> 3, \quad L-1\right)\right]$$

In case *Half_Prob* [*Channel_Nr*] = '1' the probability value is determined by:

$$P[n] = \begin{cases} 128 & : \quad 0 \leq n < N \\ T\left[\min\left(|Z[n]| >> 3, \quad L-1\right)\right] & : \quad n \geq N \end{cases}$$

This value of *P* [*n*] is applied to the arithmetic decoder that will return the value of *E* [*n*]. Next the output value *X* [*n*] (DSD sample) is the exclusive OR of *E* [*n*] and *F* [*n*]:

$$X[n] = E[n] \oplus F[n]$$

The logical value of X[n] is converted to a numerical value of Y[n] as follows:

$$Y[n] = \begin{cases} +1 & : \quad X[n] = 1 \\ -1 & : \quad X[n] = 0 \end{cases}$$

10.7.2.4 Multiplexing/Demultiplexing

The multiplexing/demultiplexing device connects each source model to the arithmetic decoder at the correct instant.

Some new variables are introduced for readability of the equations:

C N_Channels, the number of audio channels used.

n bit sequence number, range: 0 .. 8*Frame_Length – 1.

For $1 \leq i \leq C$ we have:

$$\mathrm{DST\_X\_Bit} = \mathrm{E}[0],$$

$$\mathrm{P}[0] = \mathrm{P}(\mathrm{DST\_X\_Bit}),$$

$$\mathrm{E_i}[n] = \mathrm{E}[C \cdot n + i],$$

$$\mathrm{P}[C \cdot n + i] = \mathrm{P_i}[n].$$

P(DST_X_Bit) shall be derived from Coef[0][0] (see 5.6.3.2.7) as follows. If Coef[0][0] = $\%c_8c_7c_6c_5c_4c_3c_2c_1c_0$, then P(DST_X_Bits) shall be equal to $\%0c_0c_1c_2c_3c_4c_5c_6$ + 1 of type Uint8. c_x represents the value of individual bits, with x is in the range 0..8. The value of P(DST_X_Bit) is in the range 1..128.

10.7.3 Restrictions to DST coded Audio_Frames (Normative)

To allow an optimum DST decoder design, the following restrictions must be imposed on DST coded Audio Frames.

10.7.3.1 Limited number of erroneously predicted samples

The maximum allowed number of erroneously predicted samples (see signal E in Annex C) in a DST coded Audio_Frame is half of the number of DSD samples in a Frame.

$$N\_Errors\_max = \frac{N\_Channels * Frame\_Length * 8}{2} = 18816 * N\_Channels .$$

The total number of erroneous predicted samples (N_Errors) in a Frame is the sum of the number of erroneous predicted samples per Audio Channel:

$$N\_Errors = \sum_{i=1}^{N\_Channels} \left(\sum_{n=0}^{Frame\_Length * 8 - 1} E_i[n] \right),$$

where $E_i[n]$ is either 0 (good prediction) or 1 (wrong prediction) (see Figure 10.10).

For each DST coded Audio Frame the following rule must apply:

$$N\_Errors \leq N\_Errors\_max .$$

10.7.3.2 Probability table design requirement

The restrictions defined in this paragraph must be applied to the Ptables.

For each Ptable that is used in a Frame the contents is determined with the following algorithm:

- Take all samples of the frame into account that use the Ptable in question and count for these samples how many times which Ptable entry is used (CA[ ][ ]), and how many times signal E is equal to 1 for this Ptable entry (CW[ ][ ]).

```
for (Ptable_Nr=0; Ptable_Nr<Nr_Of_Ptables; Ptable_Nr++)
{
    for (Entry_Nr=0; Entry_Nr<Ptable_Len[Ptable_Nr]; Entry_Nr++)
    {
        CA[Ptable_Nr][Entry_Nr] = 0;
        CW[Ptable_Nr][Entry_Nr] = 0;
    }
}
for (Channel_Nr=1; Channel_Nr<=N_Channels; Channel_Nr++)
{
    if (Half_Prob[Channel_Nr]==0)
    {
        Start = 0;
    }
    else
    {
        Start = Pred_Order[Filter[Channel_Nr][1]];
    }
    Stop = 0;
    for (Seg_Nr=1; Seg_Nr<=Ptables.Nr_Of_Segments[Channel_Nr]; Seg_Nr++)
    {
        Stop += 8*Ptables.Segment_Length[Channel_Nr][Seg_Nr];
        for (Bit_Nr=Start; Bit_Nr<Stop; Bit_Nr++)
        {
            Ptable_Nr = Ptable[Channel_Nr][Seg_Nr];
            Entry_Nr = min(|Z[Channel_Nr][Bit_Nr]|>>3, Ptable_Len[Ptable_Nr]-1);
            CA[Ptable_Nr][Entry_Nr]++;
            CW[Ptable_Nr][Entry_Nr] += E_Channel_Nr[Bit_Nr];
        }
        Start = Stop;
    }
}
```

- For each Entry of each Ptable the probability for wrong prediction is derived from these numbers in the following way:

```
for (Ptable_Nr=0; Ptable_Nr<Nr_Of_Ptables; Ptable_Nr++)
{
    for (Entry_Nr=0; Entry_Nr<Ptable_Len[Ptable_Nr]; Entry_Nr++)
    {
        if (CA[Ptable_Nr][Entry_Nr] == 0)
        {
            P_min[Ptable_Nr][Entry_Nr] = 1;
        }
        else
        {
            p = trunc( (512*CW[Ptable_Nr][Entry_Nr]+CA[Ptable_Nr][Entry_Nr]) / (2*CA[Ptable_Nr][Entry_Nr]) );

            P_min[Ptable_Nr][Entry_Nr] = min(max(p, 1), 128);
        }
    }
}
```

- P_min[ ][ ] are the minimum allowed probability values for the entries of the Ptables. For each Entry of each Ptable the probabilities actually used for encoding (P_one[ ][ ]) must meet the following condition:

$$P\_min[Ptable\_Nr][Entry\_Nr] \le P\_one[Ptable\_Nr][Entry\_Nr] \le 1.$$

Annex 10.A
(informative)

Encoder description

This annex describes the encoder part of the lossless audio-coding algorithm for oversampled audio signals.

10.A.1 Technical overview

As illustrated in Figure 10.A.1, in the lossless audio coding algorithm, three main processes take place; framing, prediction and entropy coding. In the encoder, the incoming one-bit data is first framed and then passed to the prediction stage. After the prediction stage, the error signal, calculated from the prediction and the original signal, is passed on to the entropy encoding stage. In this last stage the error signal is encoded using arithmetic coding, which has a performance close to the (optimal) entropy encoding. Prediction filters and probability tables can be grouped for channels for even better coding efficiency. The data generated by the arithmetic coder is finally combined with the prediction coefficients, and multiplexed in a bit-stream.

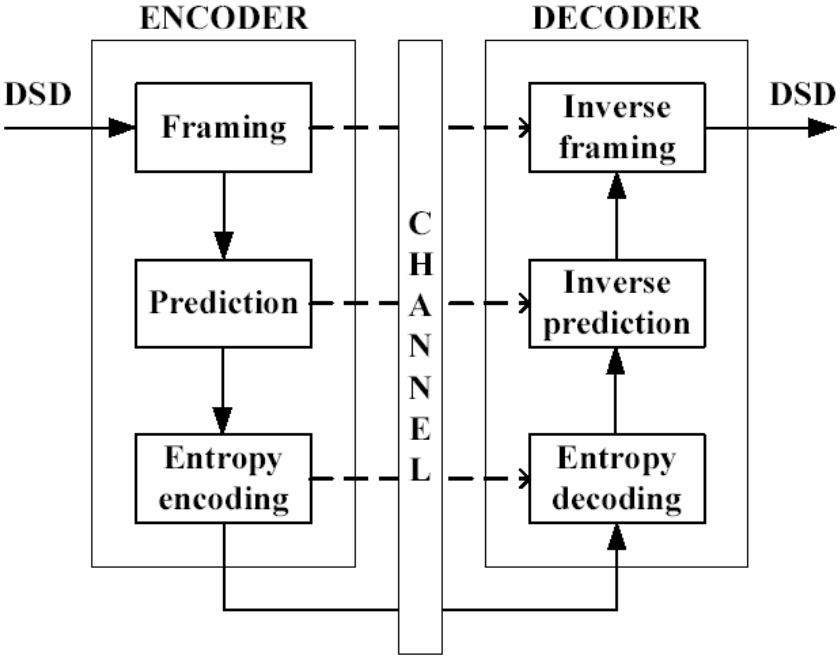


Figure 10.A.1 — Lossless Coder diagram

10.A.1.1 Framing

The purpose of framing is two-fold. Firstly, framing is necessary to provide easy, “random” access to the audio data during playback. For the same reason, each frame needs to be independently encoded, which enables the player to decode separate frames without knowledge about preceding frames. Secondly, framing allows the audio contents in a frame to be regarded as stationary (or at least, quasi-stationary). This is the underlying assumption in the prediction process.

The framing process divides the original one-bit audio stream consisting of samples $b \in \{0, 1\}$ into frames of length M=37,632 bits, corresponding to 1/75 of a second, assuming a sampling rate of 2,8224 MS/s. For other sampling rates refer to Table 10.14.

10.A.1.2 Prediction

Prediction filtering is the first necessary step in the process of (audio) data compression. The prediction filtering step, shown in more detail in Figure 10.A.2, attempts to remove redundancy from the audio bit stream b, by creating a new bit stream e, which is not redundant. Together with the prediction filter coefficients h, error stream e carries the same information as b. The prediction filter is denoted as $z^{-1}H(z)$, to emphasize the fact that the filter transfer contains a delay, which is mandatory to create an encoder that can be time-reversed (thus creating the decoder).

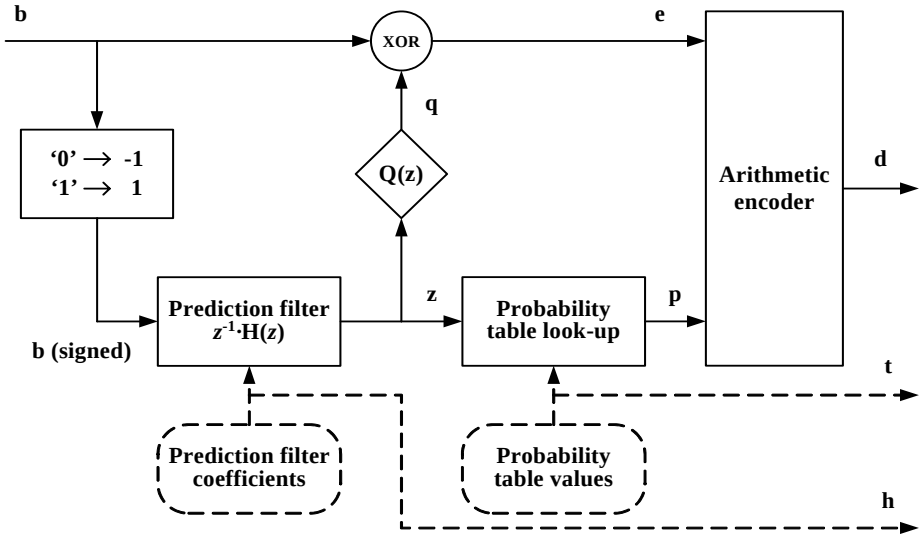


Figure 10.A.2 — Encoder block diagram

The FIR prediction filter can be designed according to standard methods, the most well-known based on Minimum Mean Squared Error (MMSE, [3]). Application of the MMSE criterion leads to the prediction error equality that needs to be minimized:

$$\sum_{n=1}^{M} \varepsilon^2[n] = \sum_{n=1}^{M} \left(\sum_{i=1}^{L} h[i] \cdot b[n-i] - b[n] \right)^2 ,$$

where M is the number of bits per frame, and L the number of prediction coefficients, which is encoded as 'Coded_Pred_Order+1'. After straightforward manipulation [1], [2], this results in the coefficients h. In general, the FIR filter found in this way will be a minimum phase filter. To obtain the optimum balance between prediction accuracy and the number of bits taken by the prediction filter description, the prediction filter coefficients are quantized to 9-bit fixed-point numbers by first scaling them by 256. The resulting coefficients are stored in 'Coef[Filter_Nr][Coef_Nr]', where Filter_Nr denotes the channel index and Coef_Nr the filter coefficient index respectively. Referring back to Figure 10.A.2, it is clear that the prediction signal z is multi-bit. The prediction bits q are derived from the multi-bit values z by simple truncation, indicated by the block labeled Q(z). It should be noted that the error between bit-stream b and multi-bit signal z is minimized, whereas ideally the difference between b and the q, the one-bit quantized version of z, would be minimized (see Figure 10.A.2). This however results in intractable mathematics.

Finally, the error signal e is calculated by an exclusive-or (XOR) operation between b and q. The purpose of the prediction filter is to create as many zeroes in e as possible, as this will enable significant data reductions by entropy encoding.

10.A.1.3 Arithmetic encoding

When proper prediction filters are used, the signal e will consist of more zeroes than ones and can thus result in a possible compression gain. Suppose that the probability of a '1' in e is denoted by p, then the probability of a '0' equals (1-p). The minimum number of bits N_{bits} with which, on average, a single bit of the stream e can be represented then equals:

$$N_{bits} = -(p \cdot \log_2(p) + (1-p) \cdot \log_2(1-p)).$$

Suppose 90% of all predictions are correct, then p = 0.1 and N_{bits} = 0.47. Typically, a compression of about a factor 2 is possible. While this calculation based on entropy calculations presents an upper limit to the achievable compression, an algorithm that under practical circumstances approaches this limit is the arithmetic-coding algorithm [3], [4].

Arithmetic encoding methods can only be used successfully when accurate information on the probabilities of the symbols “0” or “1” is available. In Figure 10.A.3, a histogram (measured over 188,160 samples taken from a 6-channel pop-recording) of the occurrence of the different values of |z| is given, indicated by diamonds. Also shown in squares, is the histogram for the occurrence of |z|, given the fact that the prediction of the sign of z is correct, *i.e.*, e=0. In triangles, the histogram is shown for the occurrence of |z|, given the fact that the prediction is wrong (e=1). These plots show that there is a very strong relationship between the value of z and the reliability of the prediction, which can be exploited in the arithmetic encoding.

The symbol probabilities needed for arithmetic coding are calculated by making a histogram (or table) as shown in Figure 10.A.3. Denoting the probability that a prediction is correct by P(e=0), we see that since P(e=0) = 1 - P(e=1), it is not necessary to calculate two tables: only the error probability table t, for P(e=1), is used for arithmetic encoding and transferred to the decoder.

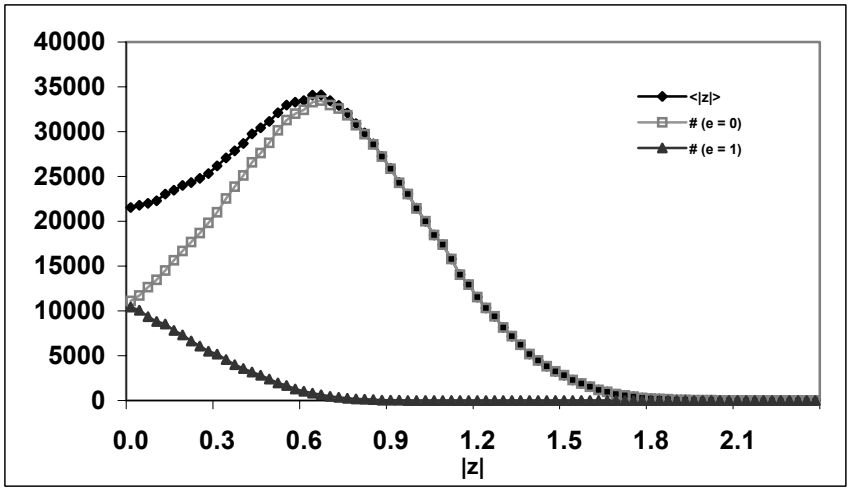


Figure 10.A.3 — Example distribution of |z| (diamonds) and the corresponding histograms for the number of right predictions (e=0, squares) and wrong predictions (e=1, triangles)

10.A.1.4 Channel multiplexing

In the previous sections the “source model” [3], consisting of the prediction filter and probability table, has been discussed for one channel. In a full encoder, every channel has its own source model, whereas only a single arithmetic encoder is used. To exploit the correlation between channels, however, it is also possible to let channels share prediction filters and/or probability tables. Sharing filters or probability tables is profitable when the decrease in number of metadata bits, necessary to transfer the filter or table information from the encoder to the decoder, is higher than the increase in number of arithmetic code bits. The latter number will typically be somewhat larger, since it is not always possible to construct a prediction filter (or probability table) that leads to optimal arithmetic encoding for all channels that are using it.

The arithmetic encoder subsequently receives, for each channel, the streams e and p, which are delivered by the individual source models.

10.A.1.5 References

[1] N.S. Jayant and P.I. Noll, .Digital Coding of Waveforms: Principles and Applications to Speech and Video., Englewood Cliffs, NJ: Prentice Hall 1984.

[2] William H. Press et al., Numerical recipes in C: the art of scientific computing., Second edition, Cambridge University Press, Cambridge, England, 1992.
[3] I.H. Witten, R.M. Neal and J.G. Cleary, “Arithmetic coding for data compression”, Communications ACM, vol. 30, pp. 520-540, June 1987.
[4] P.G. Howard and J.S. Vitter, “ Arithmetic coding for data compression”, Proc. IEEE, vol. 82, no. 6, pp. 857-865, June 1994.

ICS 35.040

Price based on 581 pages