INTERNATIONAL STANDARD ISO/IEC 14496-3:2009
TECHNICAL CORRIGENDUM 2

Published 2011-10-15

Information technology — Coding of audio-visual objects —

Part 3:
Audio

TECHNICAL CORRIGENDUM 2

Technologies de l'information — Codage des objets audio-visuels —

Partie 3: Codage audio

RECTIFICATIF TECHNIQUE 2

Technical Corrigendum 2 to ISO/IEC 14496-3:2009 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ICS 35.040 **Ref. No. ISO/IEC 14496-3:2009/Cor.2:2011(E)**

Replace Table 11.3 with:

Table 11.3 — Syntax of block_data

Syntax	No. of bits	Mnemonic
block_data()		
{		
block_type;	**1**	**uimsbf**
if (block_type == 0) {		
const_block;	**1**	**uimsbf**
js_block;	**1**	**uimsbf**
(reserved)	**5**	
if (const_block == 1) {		
{		
if (resolution == 0) { // 8 bits		
const_val;	**8**	**simsbf**
}		
else if (resolution == 1) { // 16 bits		
const_val;	**16**	**simsbf**
}		
else if (resolution == 2 || floating == 1) { // 24 bits		
const_val;	**24**	**simsbf**
}		
else { // 32 bits		
const_val;	**32**	**simsbf**
}		
}		
}		
else {		
js_block;	**1**	**uimsbf**
if ((bgmc_mode == 0) && (sb_part == 0) {		
sub_blocks = 1;		
}		
else if ((bgmc_mode == 1) && (sb_part ==1) {		
ec_sub;	**2**	**uimsbf**
sub_blocks = 1 << ec_sub;		
}		
else {		
ec_sub;	**1**	**uimsbf**
sub_blocks = (ec_sub == 1) ? 4 : 1;		
}		
if (bgmc_mode == 0) {		
for (k = 0; k < sub_blocks; k++) {		
s[k];	**varies**	**Rice code**
}		
}		
else {		
for (k = 0; k < sub_blocks; k++) {		
s[k],sx[k];	**varies**	**Rice code**
}		
}		
sb_length = block_length / sub_blocks;		
shift_lsbs;	**1**	**uimsbf**
if (shift_lsbs == 1) {		
shift_pos;	**4**	**uimsbf**
}		
if (!RLSLMS) {		

Syntax	No. of bits	Mnemonic
if (adapt_order == 1) {		
opt_order;	**1..10**	**uimsbf**
}		
else {		
opt_order = max_order;		
}		
for (p = 0; p < opt_order; p++) {		
quant_cof[p];	**varies**	**Rice code**
}		
} else {		
opt_order = 10; // for RLSLMS		
}		
if (long_term_prediction) {		
LTPenable;	**1**	**uimsbf**
if (**LTPenable**) {		
for (i = -2; i <= 2; i++) {		
LTPgain[i];	**varies**	**Rice code**
}		
LTPlag;	**8,9,10**	**uimsbf**
}		
}		
start = 0;		
if (random_access_block) {		
start = min(opt_order, min(block_length, 3));		
if (start > 0) {		
smp_val[0];	**varies**	**Rice code**
}		
if (start > 1) {		
res[1];	**varies**	**Rice code**
}		
if (start > 2) {		
res[2];	**varies**	**Rice code**
}		
}		
if (bgmc_mode) {		
for (n = start; n < sb_length; n++) {		
msb[n];	**varies**	**BGMC**
}		
for (k=1; k < sub_blocks; k++) {		
for (n = k * sb_length; n < (k+1) * sb_length; n++) {		
msb[n];	**varies**	**BGMC**
}		
}		
for (n = start; n < sb_length; n++) {		
if (msb[n] != tail_code) {		
lsb[n];	**varies**	**uimsbf**
}		
else {		
tail[n];	**varies**	**Rice code**
}		
}		
for (k=1; k < sub_blocks; k++) {		
for (n = k * sb_length; n < (k+1) * sb_length; n++) {		
if (msb[n] != tail_code) {		
lsb[n];	**varies**	**uimsbf**
}		

Syntax	No. of bits	Mnemonic
else {		
tail[n];	**varies**	**Rice code**
}		
}		
}		
}		
Else		
{		
for (n = start; n < block_length; n++) {		
res[n];	**varies**	**Rice code**
}		
}		
}		
if (RLSLMS) {		
RLSLMS_extension_data()		
}		
if (!mc_coding || js_switch) {		
byte_align;	**0..7**	**bslbf**
}		
}		

Note: random_access_block is true if the current block belongs to a random access frame (frame_id % random_access == 0) and is the first (or only) block of a channel in this frame. If non-adaptive prediction order is used (adapt_order == 0), then in random access frames the block length switching must be constrained so that no blocks in the frame need samples from the previous frame for the prediction process. The condition start <= sb_length must be true in all frames. If mc_coding is used, prohibit the use of zero block and const block (block_type == 0) as a slave channel, but permit it as a master channel. RLSLMS shall not be used together with block_switching and mc_coding.

In 11.4.3 Payloads for Floating-Point Data, replace Note after Table 11.6 Syntax of diff_float_data (changes highlighted):

Note: “byte_align” stands for padding of bits to the next byte boundary. "FlushDict()" is the function that clears and initializes the dictionary and variables of the Masked-LZ decompression module (see subclause 11.6.9).

with:

Note: “random_access_block” is defined as (random_access != 0 && (frame_id % random_access ==0)). “byte_align” stands for padding of bits to the next byte boundary. "FlushDict()" is the function that clears and initializes the dictionary and variables of the Masked-LZ decompression module (see 11.6.9).

Replace Table 11.8 with:

Table 11.8 — Syntax of Masked_LZ_decompression

Syntax	No. of bits	Mnemonic
Masked_LZ_decompression(nchars)		
{		
for (dec_chars = 0; dec_chars < nchars;) {		
string_code;	**9..15**	**uimsbf**
}		
}		

Note: “nchars” is the number of characters to be decoded (see 11.6.9).