**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).

*Replace Table 11.14 with:*

**Table 11.14 — Elements of diff_float_data**

| Field | #Bits | Description / Values |
|---|---|---|
| use_acf | 1 | 1: acf_flag[c] is present<br>0: acf_flag[c] is not present |
| acf_flag[c] | 1 | 1: acf_mantissa[c] is present<br>0: acf_mantissa[c] is not present |
| acf_mantissa[c] | 23 | Full mantissa data of common multiplier |
| highest_byte[c] | 2 | Highest nonzero bytes of mantissa in a frame |
| partA_flag[c] | 1 | 1: Samples exist in Part-A<br>0: No sample exists or all zero in Part-A |
| shift_amp[c] | 1 | 1: shift_value[c] is present<br>0: shift_value[c] is not present |
| shift_value[c] | 8 | Shift value: The shift value is biased by 127. The value (shift_value[c]-127) is added to the exponent of all floating-point values of channel c after conversion of decoded integer to floating-point values, and before addition of integer and the difference data. |

*In 11.6.9.1 Encoder for Floating-Point data, replace following sentences (changes highlighted):*

If the input signal is 32-bit floating-point, input values are decomposed as shown in 11.14 into three parts: An estimated common multiplier *A*, a truncated integer multiplicand sequence ***Y***, and a difference signal ***Z***. The same compression scheme as for normal integer input is applied for the truncated and normalized integer multiplicand sequence.

*with:*

If the input signal is 32-bit floating-point, input values are decomposed as shown in 11.14 into three parts: An estimated common multiplier *A*, a 24-bit truncated integer multiplicand sequence ***Y***, and a difference signal ***Z***. The same compression scheme as for normal 24-bit integer input is applied for the truncated and normalized integer multiplicand sequence.

*In 11.6.9.3.2.2 Normalization parameters, replace the first sentence of the second paragraph (changes highlighted):*

First, use_acd is decoded.

*with:*

First, use_acf is decoded.

*In 11.6.9.3.2.5 Masked-LZ decompression, replace the first sentence of the third paragraph (changes highlighted):*

The range of the code_bits is varied from 9 to 14 bits, since the index of the dictionary is coded as 9 to 15 bits depending on the number of the entries stored in the dictionary.

*with:*

The range of code_bits is varied from 9 to 15 bits, since the index of the dictionary is coded as 9 to 15 bits depending on the number of the entries stored in the dictionary.

*and the first sentence of the fourth paragraph:*

The deocder reads (code_bits) bits from bit stream, and gets string_code.

*with:*

The decoder reads (code_bits) bits from bit stream, and gets string_code.

*Replace 11.6.9.3.3.3 Multiplication of the common multiplier with:*

After conversion, the common multiplier *A* is reconstructed from acf_mantissa[c] and multiplied to F[c][n], and the result is set to F[c][n]. The computing procedure of multiplication is as follows.

Step 1: Sign bit setting:

The sign of the result is the same as that of F[c][n].

Step 2: Multiplication of mantissa:

(acf_mantissa[c] | 0x0800000) is multiplied to (mantissa bits of F[c][n] | 0x0800000) in a 64-bit integer resister.

Step 3: Normalization:

The result of 64-bit integer multiplication is normalized to 24-bit precision, and represented with 23 bits after discarding the top bit.

Since 1.0 <= (acf_mantissa[c] | 0x0800000)*$2^{-23}$, (mantissa part of F[c][n] | 0x0800000) *$2^{-23}$ < 2.0, the result of the multiplication is in the range [1, 4).

Consequently, it might be necessary to normalize by repeatedly shifting one bit to the right and incrementing the exponent.

Step 4: Rounding;

The rounding mode "round to nearest, to even when tie" is applied to round off the normalized mantissa of the result. Normalization process might be needed after the rounding.

*Replace 11.6.9.3.3.4 Addition of difference value of mantissa with:*

After the multiplication, the reconstructed difference value of the mantissa D[c][n] is added to the floating-point data F[c][n], and the result is set to F[c][n]. The computing procedure of addition is as follows.

Step 1: Addition of mantissa:

(D[c][n]) is added to (mantissa bits of F[c][n] | 0x0800000) in a 32-bit integer resister.

Step 2: Normalization:

The result of 32-bit integer addition is normalized to 24-bit precision, and represented with 23 bits after discarding the top bit.

Since (D[c][n])*$2^{-23}$ < 1.0, and 1.0 <= (mantissa part of F[c][n] | 0x0800000) *$2^{-23}$ < 2.0, the result of the multiplication is in the range [1, 3).

Consequently, it might be necessary to normalize by repeatedly shifting one bit to the right and incrementing the exponent.

Step 4: Truncation:

The rounding is not needed for this addition because whenever shifting occurs, the LSB of the resulting mantissa equals 0.