Information technology — Coding of audio-visual objects —
Part 3: Audio

AMENDMENT 2: Audio Lossless Coding (ALS), new audio profiles and BSAC extensions

TECHNICAL CORRIGENDUM 3

Throughout this Corrigendum, modifications or additions to existing text are highlighted in grey.

At the end of 11.2.1 Encoder and Decoder Structure, add the following sentence:

The decoder applies the inverse encoder operations in reverse order. Its output is a bit-identical version of the original input audio data.
In 11.4.3, replace:

“nbit”

with:

“nbits[c][n]”

In 11.6.9.3.2.3, 11.6.9.3.2.4, and 11.6.9.3.2.5., replace all occurrences of:

nbit[c][n]

with

nbits[c][n]

At the end of 11.6.1.4, add the following paragraph:

If MCC (Multi Channel Coding) is selected, the relationship information between channels (master or slave) is decoded. The decoded residual values of the slave channel are modified by adding those of the master channel multiplied with the decoded weighting factors (see 11.6.8). Other reconstruction processes for all channel signals, which include parameter decoding, prediction residual decoding, synthesis filtering of long-term and short-term prediction, are identical to those for decoding independent channels. The two joint channel coding tools, joint-stereo and MCC, can be adaptively selected on a frame-by-frame basis.

In 11.6.3, replace:

If the prediction order $K$ is adaptively chosen (adapt_order = 1), the number of bits used for signaling the actual order (opt_order = $K$) in each block is restricted, depending on both the global maximum order (max_order) and the block length $N_b$:

\[
\text{Bits} = \min\{\lceil\log_2(max\_order+1)\rceil, \max\{\lceil\log_2((N_b>>3)-1)\rceil, 1\}\}
\]

Therefore, also the maximum order $K_{max} = 2^{\text{Bits}} - 1$ is restricted, depending on both the value of max_order and the block length (see Table 11.19).

<table>
<thead>
<tr>
<th>$N_b$</th>
<th>max_order = 1023</th>
<th>max_order = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1023</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>511</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>255</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>127</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>63</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
If the prediction order $K$ is adaptively chosen ($\text{adapt\_order} = 1$), the number of bits used for signaling the actual order ($\text{opt\_order} = K$) in each block is restricted, depending on both the global maximum order ($\text{max\_order}$) and the block length $N_b$:

$$\text{Bits} = \min\{\lceil\log_2(\text{max\_order}+1)\rceil, \max\{\lceil\log_2(N_b\gg 3)\rceil, 1\}\}$$

Therefore, also the maximum order $K_{\text{max}} = \min\{2^{\text{Bits}} - 1, \text{max\_order}\}$ is restricted, depending on both the value of $\text{max\_order}$ and the block length (see Table 11.19).

**Table 11.19 – Examples of maximum prediction orders depending on block length and max\_order**

<table>
<thead>
<tr>
<th>$N_b$</th>
<th>$\text{max_order} = 1023$</th>
<th>$\text{max_order} = 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{#Bits for opt_order} \quad K_{\text{max}}$</td>
<td>$\text{#Bits for opt_order} \quad K_{\text{max}}$</td>
</tr>
<tr>
<td>8192</td>
<td>10 1023</td>
<td>7 100</td>
</tr>
<tr>
<td>4096</td>
<td>9 511</td>
<td>7 100</td>
</tr>
<tr>
<td>2048</td>
<td>8 255</td>
<td>7 100</td>
</tr>
<tr>
<td>1024</td>
<td>7 127</td>
<td>7 100</td>
</tr>
<tr>
<td>512</td>
<td>6 63</td>
<td>6 63</td>
</tr>
<tr>
<td>256</td>
<td>5 31</td>
<td>5 31</td>
</tr>
<tr>
<td>128</td>
<td>4 15</td>
<td>4 15</td>
</tr>
<tr>
<td>64</td>
<td>3 7</td>
<td>3 7</td>
</tr>
<tr>
<td>32</td>
<td>2 3</td>
<td>2 3</td>
</tr>
<tr>
<td>16</td>
<td>1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

**In 11.6.3.1.2, replace the first sentence:**

First, Rice-decoded residual values $\delta_k$ are combined with offsets (see Table 11.20) to produce quantized indices of parcor coefficients $a_k$:

**with:**

First, the Rice-coded residual values $\hat{\delta}_k$ are decoded and combined with offsets (see Table 11.20) to produce the quantized indices of the parcor coefficients $a_k$. 
In 11.6.9.3.2.5, replace:

The algorithm for the Masked-LZ decompression is given below.

// Masked-LZ decompression.

long n, i, readBits, string_code;
unsigned long dec_chars

last_string_code = -1;
for ( dec_chars = 0; dec_chars < nchars; ) {
    readBits = inputCode( &string_code, code_bits );
    if ( string_code == FLUSH_CODE ) || ( string_code == MAX_CODE ) ) {
        FlushDict();
    } else if ( string_code == FREEZE_CODE ) {
        freeze_flag = 1;
    } else if ( string_code == bump_code ) {
        code_bits++;
        bump_code = bump_code * 2 + 1;
    } else {
        if ( string_code >= next_code ) {
            dec_chars += decodeString( &dec_buf[dec_chars], last_string_code, &charCode );
            if ( ( dec_chars <= nchars ) && ( last_string_code != -1 ) && ( freeze_flag == 0 ) ) {
                setNewEntryToDict( next_code, last_string_code, charCode );
                next_code ++;
            }
        } else {
            dec_chars += decodeString( &dec_buf[dec_chars], last_string_code, &charCode);
            if ( ( dec_chars <= nchars ) && ( last_string_code != -1 ) && ( freeze_flag == 0 ) ) {
                setNewEntryToDict( next_code, last_string_code, charCode );
                next_code ++;
            }
        }
        last_string_code = string_code;
    }
}

Note: "dec_buf" is the buffer to store decoded characters. "nchars" is the number of characters need to be decoded. In FlushDict(), "code_bits" is set to 9, "bump_code" is set to 511 and "freeze_flag" is set to 0. After input characters are decoded form code_bits, those characters are converted into difference values of the mantissa, D[c][n].
The algorithm for the Masked-LZ decompression is given below.

// Masked-LZ decompression.

long n, i, readBits, string_code, last_string_code, charCode;
unsigned long dec_chars

last_string_code = -1;
for ( dec_chars = 0; dec_chars < nchars; ) {
    readBits = inputCode( &string_code, code_bits );
    if ( string_code == FLUSH_CODE ) || ( string_code == MAX_CODE ) {  
        FlushDict();
        last_string_code = -1;
    }  
    else if ( string_code == FREEZE_CODE ) {  
        freeze_flag = 1;
    }  
    else if ( string_code == bump_code ) {  
        code_bits++;
        bump_code = bump_code * 2 + 1;
    }  
    else {  
        if ( string_code >= next_code ) {  
            dec_chars += decodeString( &dec_buf[dec_chars], last_string_code, &charCode );
            dec_chars += decodeString( &dec_buf[dec_chars], charCode, &charCode );
            setNewEntryToDict( next_code, last_string_code, charCode );
            next_code ++;
        }  
        else {  
            dec_chars += decodeString(&dec_buf[dec_chars], string_code, &charCode);
            if ( ( dec_chars < nchars ) && ( last_string_code != -1 ) && ( freeze_flag == 0 ) ) {  
                setNewEntryToDict( next_code, last_string_code, charCode );
                next_code ++;
            }
        }
    }
    last_string_code = string_code;
}

Note: The inputCode() function reads the "code_bits" number of bits from the encoded bitstream and returns a string_code. The decodeString() function takes a string_code as an input value and returns the decoded character string related to the string_code by searching the dictionary, the number of characters in the decoded string and a code of the first character of the string, charCode. The setNewEntryToDict() function takes a last_string_code and a charCode, and sets them to a vacant entry of the dictionary represented as the next_code. The FlushDict() function clears all entries of the dictionary and initializes the related values of the dictionary. In the above pseudo code, "dec_buf" is the buffer to store decoded characters, and "nchars" is the number of characters that need to be decoded. In FlushDict(), "code_bits" is set to 9, "bump_code" is set to 511 and "freeze_flag" is set to 0. After input characters are decoded from string_code, those characters are converted into difference values of the mantissa, D[c][n].
In 11.6.9.3.2.5, replace:

When \( nbits[c][n] \) is not the multiple of 8 which is the word size of characters used in Masked-LZ module, it means that dummy bits are added at the encoder side.

Additional bits longer than the \( nbits[c][n] \) are cut off (thrown away) using following algorithm:

```c
// reconstruction of difference values from decoded characters.
long n, i, nbits_aligned;
unsigned long acc, j;
j = 0;
for (n = 0; n < frame_length; n++) {
    if ( !int_zero[c][n] ) {
        if ( nbits[c][n] % 8 ) > 0 )
            nbits_aligned = 8 * (unsigned int)(nbits[c][n] / 8) + 1;
        else
            nbits_aligned = nbits[c][n];
        acc = 0;
        for ( i = 0; i < nbits_aligned / 8; i++ )
            acc = ( acc << 8 ) + dec_buf[j++];
        acc >>= ( nbits_aligned - nbits[c][n] );  // throw away dummy bits added by the encoder.
        D[c][n] = acc;
    }
}
```

with:

If \( nbits[c][n] \) is not a multiple of 8, which is the word size of characters used in the Masked-LZ module, this means that dummy bits were added at the encoder side.

Additional bits longer than the \( nbits[c][n] \) are cut off (thrown away) using the following algorithm:

```c
// reconstruction of difference values from decoded characters.
long n, i, nbits_aligned;
unsigned long acc, j;
j = 0;
for (n = 0; n < frame_length; n++) {
    if ( !int_zero[c][n] ) {
        if ( nbits[c][n] % 8 ) > 0 )
            nbits_aligned = 8 * ((unsigned int)(nbits[c][n] / 8) + 1);
        else
            nbits_aligned = nbits[c][n];
        acc = 0;
        for ( i = 0; i < nbits_aligned / 8; i++ )
            acc = ( acc << 8 ) + dec_buf[j++];
        acc >>= ( nbits_aligned - nbits[c][n] );  // throw away dummy bits added by the encoder.
        D[c][n] = acc;
    }
}
```