INTERNATIONAL STANDARD ISO/IEC 14496-3:2005/Amd.3:2006
TECHNICAL CORRIGENDUM 1

Published 2008-12-01

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION
INTERNATIONAL ELECTROTECHNICAL COMMISSION • МЕЖДУНАРОДНАЯ ЭЛЕКТРОТЕХНИЧЕСКАЯ КОМИССИЯ • COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

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

AMENDMENT 3: Scalable Lossless Coding (SLS)

TECHNICAL CORRIGENDUM 1

Technologies de l'information — Codage des objets audiovisuels —

Partie 3: Codage audio

AMENDEMENT 3: Codage extensible sans perte (SLS)

RECTIFICATIF TECHNIQUE 1

Technical Corrigendum 1 to ISO/IEC 14496-3:2005/Amd.3:2006 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

Replace 12.5.5.2.1 entirely, with the following (where differences to the existing text are highlighted in grey):

12.5.5.2.1 Overview

The residual integer spectral data vector is decoded from the LLE data stream *lle_data*(). Firstly, all scale factor bands with *lazy_bp* > 0 are BPGC/CBAC decoded, where the amplitude and sign of the residual spectral data *res* is bit-plane decoded starting from the maximum bit-plane *max_bp* and progressing to lower bit-planes until bit-plane 0 for each scale factor band. Subsequently, the low energy mode decoding is invoked to decode the remaining scale factor bands with *lazy_bp* <= 0.

ICS 35.040 **Ref. No. ISO/IEC 14496-3:2005/Amd.3:2006/Cor.1:2008(E)**

Published in Switzerland

The SLS decoder can provide the functionality of fine-grain scalability (FGS) by truncating the LLE bitstream. Moreover, it allows to decode additional symbols beyond the point of truncation by exploiting the properties of arithmetic coding.

In 12.5.5.2.2 replace the text up to Table 12.19 with the following (where differences to the existing text are highlighted in grey):

12.5.5.2.2 BPGC/CBAC decoding process

The BPGC decoding or CBAC decoding process is performed on scale factor bands for which *lazy_bp*>0. The BPGC/CBAC bit-plane decoding process is used to decode the bit-plane symbols for reconstructing the residual integer spectral data *res*. The bit-plane decoding process is started from *max_bp* for each sfb, and progressively proceeds to lower bit-planes. For the first NUM_BP bit-plane scans the bit-plane symbols are arithmetic decoded as illustrated in the following pseudo code:

```
/* preparing the help element */
for (g=0;g<num_window_groups;g++){
    for (sfb = 0;sfb<num_sfb;sfb++){
        width = swb_offset[g][sfb+1] - swb_offset[g][sfb];
        for (win = 0;win <window_group_len[g];win++) {
            for (bin=0;bin<width;bin++){
                is_sig[g][win][sfb][bin] =
                (quant[g][sfb][win][bin])&&(band_type[g][sfb]==ImplicitBand)?1:0;
                /* sign will be determined implicitly if is_sig == 1 */
                amp[g][win][sfb][bin] = 0;
                sign[g][win][sfb][bin] = 1;
            }
        }
        cur_bp[g][sfb] = max_bp[g][sfb];
    }
}

/* BPGC/CBAC decoding */
while ((max_bp[g][sfb] - cur_bp[g][sfb]<LAZY_BP) && (cur_bp[g][sfb] >= 0)){
    for (g=0;g<num_window_groups;g++){
        for (sfb = 0;sfb<num_sfb;sfb++){
            if ((cur_bp[g][sfb]>=0) && (lazy_bp[g][sfb] > 0)){
                width = swb_offset[g][sfb+1] - swb_offset[g][sfb];
                for (win=0;win<window_group_len[g];win++){
                    for (bin=0;bin<width;bin++){
                        if (!is_lle_ics_eof ()){
                            if (interval[g][win][sfb][bin] >
                                amp[g][win][sfb][bin] + (1<<cur_bp[g][sfb])
                            {
                                freq = determine_frequency();
                                sym = decode(freq);
                                amp[g][win][sfb][bin] += sym << cur_bp[g][sfb];
                                 /* decode bit-plane cur_bp*/
                                if ((!is_sig[g][win][sfb][bin]) && (sym )) {
                                        /* decode sign bit of res if necessary */
                                        sign[g][win][sfb][bin] = (decode(freq_sign))? -1:1;
                                        is_sig[g][win][sfb][bin] = 1;

                                }
                            }
                        }
                        else {
                            smart_decoding_cbac_bpgc();
                        }
                    }
                }
                cur_bp[g][sfb]--; /* progress to next bit-plane */
            }
        }
    }
}
```

After that, BPGC/CBAC enters the lazy decoding mode after skipping the 2 bit terminating string, where the bit-plane symbols are directly read from the input bit-stream:

```
/* BPGC/CBAC lazy decoding */
read_bits(1);read_bits(1); /* skip the 2 AC termination string before lazy coding
while (cur_bp[g][sfb] >= 0){
    for (g=0;g<num_window_groups;g++){
        for (sfb = 0;sfb<num_sfb+num_osf_sfb;sfb++){
            if ((cur_bp[g][sfb]>=0) && (lazy_bp[g][sfb] > 0)){
                width = swb_offset[g][sfb+1] - swb_offset[g][sfb];
                for (win=0;win<window_group_len[g];win++){
                    for (bin=0;bin<width;bin++){
                        if (!is_lle_ics_eof ()){
                            if (interval[g][win][sfb][bin] >
                                amp[g][win][sfb][bin] + (1<<cur_bp[g][sfb])
                            {
                                sym = read_bit();
                                amp[g][win][sfb][bin] += sym << cur_bp[g][sfb];
                                 /* decode bit-plane cur_bp */
                                if ((!is_sig[g][win][sfb][bin]) && (sym )) {
                                        /* decode sign bit of res if necessary */
                                        sign[g][win][sfb][bin] = (decode(freq_sign))? -1:1;
                                        is_sig[g][win][sfb][bin] = 1;
                                }
                            }
                        }
                    }
                }
                cur_bp[g][sfb]--;
            }
        }
    }
}
```

The value of NUM_BP is determined in the following table.

In 12.5.5.2.3 replace the text up to Table 12.25 with the following (where differences to the existing text are highlighted in grey):

12.5.5.2.3 Low Energy Mode Code (LEMC) Decoding

The following pseudo code illustrates the LEMC decoding process that is performed on scale factor bands for which *lazy*_bp<=0.

```
/* low energy mode decoding */
for (g = 0;g < num_window_groups; g++){
    for (sfb = 0; sfb <num_sfb+num_osf_sfb;sfb++){
        if ((cur_bp[g][sfb] >= 0) && (lazy_bp[g][sfb] <= 0))
        {
            width = swb_offset[g][sfb+1] - swb_offset[g][sfb];
            for (win=0;win<window_group_len[g];win++){
                sym = 0;
                pos = 0;
                for (bin=0;bin<width;bin++){
                    if (!is_lle_ics_eof ()){
                        /* decoding of binary string and reconstructing res */
                        while (decode(freq_silence[pos])==1) {
                            sym++;
                            pos++;
                            if (pos>2) pos = 2;
                            if (sym==(1<<(max_bp[g][sfb]+1))-1) break;
                        }
                        amp[g][sfb][win][bin]+=sym;
                        /* decoding of sign of res */
                        if ((!is_sig[g][win][sfb][bin]) && (sym )) {
```

```
    ````                        sign[g][win][sfb][bin] = (decode(freq_sign))? -1:1;
                            is_sig[g][win][sfb][bin] = 1;
                        }
                    }
                    else smart_decoding_low_energy();
                }
            }
        }
    }
}
```

The probability assignments for the low energy mode decoding, *freq_bpgc* and *freq_silence* are given in the following tables. The sign bits in the above decoding process are decoded with frequency 8192, i.e. *freq_sign* = 8192.

Replace 12.5.5.2.5 entirely, with the following (where differences to the existing text are highlighted in grey):

12.5.5.2.5 Smart Arithmetic Decoding of Truncated SLS Bitstreams

The smart arithmetic decoding provides an efficient way to decode an intermediate layer corresponding to a given target bitrate. This algorithm exploits the fact that a decoding buffer still contains meaningful information for arithmetic decoding even if there is no bit left to be fed into the decoding buffer. The decoding process continues as long as there exists no ambiguity in determining a symbol.
The following pseudo code illustrates the algorithm to detect the ambiguity in the arithmetic decoding module. The variable *num_dummy_bits* represents the number of calls to evoke the function of *read_bits(1)* in the arithmetic decoding process just after the truncation point.

```
int ambiguity_check(int freq)
{
    /* if there is no ambiguity, returns 1 */
    /* otherwise, returns 0                */
    upper = 1<<num_dummy_bits;
    decisionVal = ((high-low)*freq>>PRE_SHT)-value+low-1;
    if(decisionVal>upper || decisionVal<0) return 0;
    else return 1;
}
```

Either *smart_decoding_cbac_bpgc()* or *smart_decoding_low_energy()* is executed when *num_dummy_bits* is greater than 0. In order to prevent sign bit errors, the spectral value of the current spectral line should be set to zero when an ambiguity can occur while decoding a sign bit. Notice that all index variables in the smart decoding process should be carried over from the previous arithmetic decoding process.

```
smart_decoding_cbac_bpgc()
{
    /* BPGC/CBAC normal decoding with ambiguity detection */
    while ((max_bp[g][sfb] - cur_bp[g][sfb]<LAZY_BP) && (cur_bp[g][sfb] >= 0)){
        for (;g<num_window_groups;g++){
            for (;sfb<num_sfb;sfb++){
                if ((cur_bp[g][sfb]>=0) && (lazy_bp[g][sfb] > 0)){
                    width = swb_offset[g][sfb+1] - swb_offset[g][sfb];
                    for (;win<window_group_len[g];win++){
                        for (;bin<width;bin++){
                            if (interval[g][win][sfb][bin] >
                                amp[g][win][sfb][bin] + (1<<cur_bp[g][sfb])
                            {
                                freq = determine_frequency();
                                if (ambiguity_check(freq)) {
                                        /* no ambiguity for arithmetic decoding */
                                        sym = decode(freq);
                                        amp[g][win][sfb][bin] += sym << cur_bp[g][sfb];
                                        /* decode bit-plane cur_bp*/
                                        if ((!is_sig[g][win][sfb][bin]) && (sym)) {
                                                /* decode sign bit of res if necessary */
```
```