```h
#ifndef MINIMP3_H
#define MINIMP3_H
/*
    https://github.com/lieff/minimp3
    To the extent possible under law, the author(s) have dedicated all copyright and related and neighboring rights to this software to the public domain worldwide.
    This software is distributed without any warranty.
    See <http://creativecommons.org/publicdomain/zero/1.0/>.
*/
#include <stdint.h>

#define MINIMP3_MAX_SAMPLES_PER_FRAME (1152 * 2)

typedef struct
{
  int frame_bytes, frame_offset, channels, hz, layer, bitrate_kbps;
} mp3dec_frame_info_t;

typedef struct
{
  float mdct_overlap[2][9 * 32], qmf_state[15 * 2 * 32];
  int reserv, free_format_bytes;
  unsigned char header[4], reserv_buf[511];
} mp3dec_t;

#ifdef __cplusplus
extern "C"
{
#endif /* __cplusplus */

  void mp3dec_init(mp3dec_t *dec);
#ifndef MINIMP3_FLOAT_OUTPUT
  typedef int16_t mp3d_sample_t;
#else  /* MINIMP3_FLOAT_OUTPUT */
typedef float mp3d_sample_t;
void mp3dec_f32_to_s16(const float *in, int16_t *out, int num_samples);
#endif /* MINIMP3_FLOAT_OUTPUT */
  int mp3dec_decode_frame(mp3dec_t *dec, const uint8_t *mp3, int mp3_bytes, mp3d_sample_t *pcm, mp3dec_frame_info_t *info);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* MINIMP3_H */
#if defined(MINIMP3_IMPLEMENTATION) && !defined(_MINIMP3_IMPLEMENTATION_GUARD)
#define _MINIMP3_IMPLEMENTATION_GUARD

#include <stdlib.h>
#include <string.h>
```

```
49
50 #define MAX_FREE_FORMAT_FRAME_SIZE 2304 /* more than ISO spec's */
51 #ifndef MAX_FRAME_SYNC_MATCHES
52 #define MAX_FRAME_SYNC_MATCHES 10
53 #endif /* MAX_FRAME_SYNC_MATCHES */
54
55 #define MAX_L3_FRAME_PAYLOAD_BYTES MAX_FREE_FORMAT_FRAME_SIZE /* MUST
     be >= 320000/8/32000*1152 = 1440 */
56
57 #define MAX_BITRESERVOIR_BYTES 511
58 #define SHORT_BLOCK_TYPE 2
59 #define STOP_BLOCK_TYPE 3
60 #define MODE_MONO 3
61 #define MODE_JOINT_STEREO 1
62 #define HDR_SIZE 4
63 #define HDR_IS_MONO(h) (((h[3]) & 0xC0) == 0xC0)
64 #define HDR_IS_MS_STEREO(h) (((h[3]) & 0xE0) == 0x60)
65 #define HDR_IS_FREE_FORMAT(h) (((h[2]) & 0xF0) == 0)
66 #define HDR_IS_CRC(h) (!((h[1]) & 1))
67 #define HDR_TEST_PADDING(h) ((h[2]) & 0x2)
68 #define HDR_TEST_MPEG1(h) ((h[1]) & 0x8)
69 #define HDR_TEST_NOT_MPEG25(h) ((h[1]) & 0x10)
70 #define HDR_TEST_I_STEREO(h) ((h[3]) & 0x10)
71 #define HDR_TEST_MS_STEREO(h) ((h[3]) & 0x20)
72 #define HDR_GET_STEREO_MODE(h) (((h[3]) >> 6) & 3)
73 #define HDR_GET_STEREO_MODE_EXT(h) (((h[3]) >> 4) & 3)
74 #define HDR_GET_LAYER(h) (((h[1]) >> 1) & 3)
75 #define HDR_GET_BITRATE(h) ((h[2]) >> 4)
76 #define HDR_GET_SAMPLE_RATE(h) (((h[2]) >> 2) & 3)
77 #define HDR_GET_MY_SAMPLE_RATE(h) (HDR_GET_SAMPLE_RATE(h) + (((h[1] >>
      3) & 1) + ((h[1] >> 4) & 1)) * 3)
78 #define HDR_IS_FRAME_576(h) ((h[1] & 14) == 2)
79 #define HDR_IS_LAYER_1(h) ((h[1] & 6) == 6)
80
81 #define BITS_DEQUANTIZER_OUT -1
82 #define MAX_SCF (255 + BITS_DEQUANTIZER_OUT * 4 - 210)
83 #define MAX_SCFI ((MAX_SCF + 3) & ~3)
84
85 #define MINIMP3_MIN(a, b) ((a) > (b) ? (b) : (a))
86 #define MINIMP3_MAX(a, b) ((a) < (b) ? (b) : (a))
87
88 #if !defined(MINIMP3_NO_SIMD)
89
90 #if !defined(MINIMP3_ONLY_SIMD) && (defined(_M_X64) || defined
     (__x86_64__) || defined(__aarch64__) || defined(_M_ARM64))
91 /* x64 always have SSE2, arm64 always have neon, no need for generic
     code */
92 #define MINIMP3_ONLY_SIMD
93 #endif /* SIMD checks... */
94
95 #if (defined(_MSC_VER) && (defined(_M_IX86) || defined(_M_X64))) ||
     ((defined(__i386__) || defined(__x86_64__)) && defined(__SSE2__))
96 #if defined(_MSC_VER)
```

```
 97 #include <intrin.h>
 98 #endif /* defined(_MSC_VER) */
 99 #include <immintrin.h>
100 #define HAVE_SSE 1
101 #define HAVE_SIMD 1
102 #define VSTORE _mm_storeu_ps
103 #define VLD _mm_loadu_ps
104 #define VSET _mm_set1_ps
105 #define VADD _mm_add_ps
106 #define VSUB _mm_sub_ps
107 #define VMUL _mm_mul_ps
108 #define VMAC(a, x, y) _mm_add_ps(a, _mm_mul_ps(x, y))
109 #define VMSB(a, x, y) _mm_sub_ps(a, _mm_mul_ps(x, y))
110 #define VMUL_S(x, s) _mm_mul_ps(x, _mm_set1_ps(s))
111 #define VREV(x) _mm_shuffle_ps(x, x, _MM_SHUFFLE(0, 1, 2, 3))
112 typedef __m128 f4;
113 #if defined(_MSC_VER) || defined(MINIMP3_ONLY_SIMD)
114 #define minimp3_cpuid __cpuid
115 #else /* defined(_MSC_VER) || defined(MINIMP3_ONLY_SIMD) */
116 static __inline__ __attribute__((always_inline)) void minimp3_cpuid
      (int CPUInfo[], const int InfoType)
117 {
118 #if defined(__PIC__)
119   __asm__ __volatile__(
120 #if defined(__x86_64__)
121       "push %%rbx\n"
122       "cpuid\n"
123       "xchgl %%ebx, %1\n"
124       "pop  %%rbx\n"
125 #else  /* defined(__x86_64__) */
126       "xchgl %%ebx, %1\n"
127       "cpuid\n"
128       "xchgl %%ebx, %1\n"
129 #endif /* defined(__x86_64__) */
130       : "=a"(CPUInfo[0]), "=r"(CPUInfo[1]), "=c"(CPUInfo[2]),
          "=d"(CPUInfo[3])
131       : "a"(InfoType));
132 #else  /* defined(__PIC__) */
133   __asm__ __volatile__(
134       "cpuid"
135       : "=a"(CPUInfo[0]), "=b"(CPUInfo[1]), "=c"(CPUInfo[2]),
          "=d"(CPUInfo[3])
136       : "a"(InfoType));
137 #endif /* defined(__PIC__)*/
138 }
139 #endif /* defined(_MSC_VER) || defined(MINIMP3_ONLY_SIMD) */
140 static int have_simd(void)
141 {
142 #ifdef MINIMP3_ONLY_SIMD
143   return 1;
144 #else /* MINIMP3_ONLY_SIMD */
145   static int g_have_simd;
146   int CPUInfo[4];
```

```
147 #ifdef MINIMP3_TEST
148   static int g_counter;
149   if (g_counter++ > 100)
150     return 0;
151 #endif /* MINIMP3_TEST */
152   if (g_have_simd)
153     goto end;
154   minimp3_cpuid(CPUInfo, 0);
155   g_have_simd = 1;
156   if (CPUInfo[0] > 0)
157   {
158     minimp3_cpuid(CPUInfo, 1);
159     g_have_simd = (CPUInfo[3] & (1 << 26)) + 1; /* SSE2 */
160   }
161 end:
162   return g_have_simd - 1;
163 #endif /* MINIMP3_ONLY_SIMD */
164 }
165 #elif defined(__ARM_NEON) || defined(__aarch64__) || defined(_M_ARM64)
166 #include <arm_neon.h>
167 #define HAVE_SSE 0
168 #define HAVE_SIMD 1
169 #define VSTORE vst1q_f32
170 #define VLD vld1q_f32
171 #define VSET vmovq_n_f32
172 #define VADD vaddq_f32
173 #define VSUB vsubq_f32
174 #define VMUL vmulq_f32
175 #define VMAC(a, x, y) vmlaq_f32(a, x, y)
176 #define VMSB(a, x, y) vmlsq_f32(a, x, y)
177 #define VMUL_S(x, s) vmulq_f32(x, vmovq_n_f32(s))
178 #define VREV(x) vcombine_f32(vget_high_f32(vrev64q_f32(x)),
      vget_low_f32(vrev64q_f32(x)))
179 typedef float32x4_t f4;
180 static int have_simd()
181 { /* TODO: detect neon for !MINIMP3_ONLY_SIMD */
182   return 1;
183 }
184 #else /* SIMD checks... */
185 #define HAVE_SSE 0
186 #define HAVE_SIMD 0
187 #ifdef MINIMP3_ONLY_SIMD
188 #error MINIMP3_ONLY_SIMD used, but SSE/NEON not enabled
189 #endif /* MINIMP3_ONLY_SIMD */
190 #endif /* SIMD checks... */
191 #else  /* !defined(MINIMP3_NO_SIMD) */
192 #define HAVE_SIMD 0
193 #endif /* !defined(MINIMP3_NO_SIMD) */
194
195 #if defined(__ARM_ARCH) && (__ARM_ARCH >= 6) && !defined(__aarch64__)
      && !defined(_M_ARM64)
196 #define HAVE_ARMV6 1
197 static __inline__ __attribute__((always_inline)) int32_t
```

```
    minimp3_clip_int16_arm(int32_t a)
198 {
199   int32_t x = 0;
200   __asm__("ssat %0, #16, %1"
201           : "=r"(x)
202           : "r"(a));
203   return x;
204 }
205 #else
206 #define HAVE_ARMV6 0
207 #endif
208
209 typedef struct
210 {
211   const uint8_t *buf;
212   int pos, limit;
213 } bs_t;
214
215 typedef struct
216 {
217   float scf[3 * 64];
218   uint8_t total_bands, stereo_bands, bitalloc[64], scfcod[64];
219 } L12_scale_info;
220
221 typedef struct
222 {
223   uint8_t tab_offset, code_tab_width, band_count;
224 } L12_subband_alloc_t;
225
226 typedef struct
227 {
228   const uint8_t *sfbtab;
229   uint16_t part_23_length, big_values, scalefac_compress;
230   uint8_t global_gain, block_type, mixed_block_flag, n_long_sfb,       ↵
      n_short_sfb;
231   uint8_t table_select[3], region_count[3], subblock_gain[3];
232   uint8_t preflag, scalefac_scale, count1_table, scfsi;
233 } L3_gr_info_t;
234
235 typedef struct
236 {
237   bs_t bs;
238   uint8_t maindata[MAX_BITRESERVOIR_BYTES +                          ↵
      MAX_L3_FRAME_PAYLOAD_BYTES];
239   L3_gr_info_t gr_info[4];
240   float grbuf[2][576], scf[40], syn[18 + 15][2 * 32];
241   uint8_t ist_pos[2][39];
242 } mp3dec_scratch_t;
243
244 static void bs_init(bs_t *bs, const uint8_t *data, int bytes)
245 {
246   bs->buf = data;
247   bs->pos = 0;
```

```
248   bs->limit = bytes * 8;
249 }
250
251 static uint32_t get_bits(bs_t *bs, int n)
252 {
253   uint32_t next, cache = 0, s = bs->pos & 7;
254   int shl = n + s;
255   const uint8_t *p = bs->buf + (bs->pos >> 3);
256   if ((bs->pos += n) > bs->limit)
257     return 0;
258   next = *p++ & (255 >> s);
259   while ((shl -= 8) > 0)
260   {
261     cache |= next << shl;
262     next = *p++;
263   }
264   return cache | (next >> -shl);
265 }
266
267 static int hdr_valid(const uint8_t *h)
268 {
269   return h[0] == 0xff &&
270          ((h[1] & 0xF0) == 0xf0 || (h[1] & 0xFE) == 0xe2) &&
271          (HDR_GET_LAYER(h) != 0) &&
272          (HDR_GET_BITRATE(h) != 15) &&
273          (HDR_GET_SAMPLE_RATE(h) != 3);
274 }
275
276 static int hdr_compare(const uint8_t *h1, const uint8_t *h2)
277 {
278   return hdr_valid(h2) &&
279          ((h1[1] ^ h2[1]) & 0xFE) == 0 &&
280          ((h1[2] ^ h2[2]) & 0x0C) == 0 &&
281          !(HDR_IS_FREE_FORMAT(h1) ^ HDR_IS_FREE_FORMAT(h2));
282 }
283
284 static unsigned hdr_bitrate_kbps(const uint8_t *h)
285 {
286   static const uint8_t halfrate[2][3][15] = {
287       {{0, 4, 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, 64, 72, 80}, {0,
          4, 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, 64, 72, 80}, {0, 16,
          24, 28, 32, 40, 48, 56, 64, 72, 80, 88, 96, 112, 128}},
288       {{0, 16, 20, 24, 28, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160},
           {0, 16, 24, 28, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160,
          192}, {0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192,
          208, 224}},
289   };
290   return 2 * halfrate[!!HDR_TEST_MPEG1(h)][HDR_GET_LAYER(h) - 1]
        [HDR_GET_BITRATE(h)];
291 }
292
293 static unsigned hdr_sample_rate_hz(const uint8_t *h)
294 {
```

```
295   static const unsigned g_hz[3] = {44100, 48000, 32000};
296   return g_hz[HDR_GET_SAMPLE_RATE(h)] >> (int)!HDR_TEST_MPEG1(h) >>
        (int)!HDR_TEST_NOT_MPEG25(h);
297 }
298
299 static unsigned hdr_frame_samples(const uint8_t *h)
300 {
301   return HDR_IS_LAYER_1(h) ? 384 : (1152 >> (int)HDR_IS_FRAME_576(h));
302 }
303
304 static int hdr_frame_bytes(const uint8_t *h, int free_format_size)
305 {
306   int frame_bytes = hdr_frame_samples(h) * hdr_bitrate_kbps(h) * 125 /
        hdr_sample_rate_hz(h);
307   if (HDR_IS_LAYER_1(h))
308   {
309     frame_bytes &= ~3; /* slot align */
310   }
311   return frame_bytes ? frame_bytes : free_format_size;
312 }
313
314 static int hdr_padding(const uint8_t *h)
315 {
316   return HDR_TEST_PADDING(h) ? (HDR_IS_LAYER_1(h) ? 4 : 1) : 0;
317 }
318
319 #ifndef MINIMP3_ONLY_MP3
320 static const L12_subband_alloc_t *L12_subband_alloc_table(const
      uint8_t *hdr, L12_scale_info *sci)
321 {
322   const L12_subband_alloc_t *alloc;
323   int mode = HDR_GET_STEREO_MODE(hdr);
324   int nbands, stereo_bands = (mode == MODE_MONO) ? 0 : (mode ==
        MODE_JOINT_STEREO) ? (HDR_GET_STEREO_MODE_EXT(hdr) << 2) + 4
325                : 32;
326
327   if (HDR_IS_LAYER_1(hdr))
328   {
329     static const L12_subband_alloc_t g_alloc_L1[] = {{76, 4, 32}};
330     alloc = g_alloc_L1;
331     nbands = 32;
332   }
333   else if (!HDR_TEST_MPEG1(hdr))
334   {
335     static const L12_subband_alloc_t g_alloc_L2M2[] = {{60, 4, 4},
        {44, 3, 7}, {44, 2, 19}};
336     alloc = g_alloc_L2M2;
337     nbands = 30;
338   }
339   else
340   {
341     static const L12_subband_alloc_t g_alloc_L2M1[] = {{0, 4, 3}, {16,
```

```
          4, 8}, {32, 3, 12}, {40, 2, 7}};
342     int sample_rate_idx = HDR_GET_SAMPLE_RATE(hdr);
343     unsigned kbps = hdr_bitrate_kbps(hdr) >> (int)(mode != MODE_MONO);
344     if (!kbps) /* free-format */
345     {
346       kbps = 192;
347     }
348
349     alloc = g_alloc_L2M1;
350     nbands = 27;
351     if (kbps < 56)
352     {
353       static const L12_subband_alloc_t g_alloc_L2M1_lowrate[] = {{44,
          4, 2}, {44, 3, 10}};
354       alloc = g_alloc_L2M1_lowrate;
355       nbands = sample_rate_idx == 2 ? 12 : 8;
356     }
357     else if (kbps >= 96 && sample_rate_idx != 1)
358     {
359       nbands = 30;
360     }
361   }
362
363   sci->total_bands = (uint8_t)nbands;
364   sci->stereo_bands = (uint8_t)MINIMP3_MIN(stereo_bands, nbands);
365
366   return alloc;
367 }
368
369 static void L12_read_scalefactors(bs_t *bs, uint8_t *pba, uint8_t
      *scfcod, int bands, float *scf)
370 {
371   static const float g_deq_L12[18 * 3] = {
372 #define DQ(x) 9.53674316e-07f / x, 7.56931807e-07f / x,
      6.00777173e-07f / x
373       DQ(3), DQ(7), DQ(15), DQ(31), DQ(63), DQ(127), DQ(255), DQ(511),
          DQ(1023), DQ(2047), DQ(4095), DQ(8191), DQ(16383), DQ(32767),
         DQ(65535), DQ(3), DQ(5), DQ(9)};
374   int i, m;
375   for (i = 0; i < bands; i++)
376   {
377     float s = 0;
378     int ba = *pba++;
379     int mask = ba ? 4 + ((19 >> scfcod[i]) & 3) : 0;
380     for (m = 4; m; m >>= 1)
381     {
382       if (mask & m)
383       {
384         int b = get_bits(bs, 6);
385         s = g_deq_L12[ba * 3 - 6 + b % 3] * (1 << 21 >> b / 3);
386       }
387       *scf++ = s;
388     }
```

```
389     }
390   }
391
392   static void L12_read_scale_info(const uint8_t *hdr, bs_t *bs,
        L12_scale_info *sci)
393   {
394     static const uint8_t g_bitalloc_code_tab[] = {
395         0, 17, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
396         0, 17, 18, 3, 19, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16,
397         0, 17, 18, 3, 19, 4, 5, 16,
398         0, 17, 18, 16,
399         0, 17, 18, 19, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
400         0, 17, 18, 3, 19, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
401         0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
402     const L12_subband_alloc_t *subband_alloc = L12_subband_alloc_table
        (hdr, sci);
403
404     int i, k = 0, ba_bits = 0;
405     const uint8_t *ba_code_tab = g_bitalloc_code_tab;
406
407     for (i = 0; i < sci->total_bands; i++)
408     {
409       uint8_t ba;
410       if (i == k)
411       {
412         k += subband_alloc->band_count;
413         ba_bits = subband_alloc->code_tab_width;
414         ba_code_tab = g_bitalloc_code_tab + subband_alloc->tab_offset;
415         subband_alloc++;
416       }
417       ba = ba_code_tab[get_bits(bs, ba_bits)];
418       sci->bitalloc[2 * i] = ba;
419       if (i < sci->stereo_bands)
420       {
421         ba = ba_code_tab[get_bits(bs, ba_bits)];
422       }
423       sci->bitalloc[2 * i + 1] = sci->stereo_bands ? ba : 0;
424     }
425
426     for (i = 0; i < 2 * sci->total_bands; i++)
427     {
428       sci->scfcod[i] = sci->bitalloc[i] ? HDR_IS_LAYER_1(hdr) ? 2 :
          get_bits(bs, 2) : 6;
429     }
430
431     L12_read_scalefactors(bs, sci->bitalloc, sci->scfcod, sci-
        >total_bands * 2, sci->scf);
432
433     for (i = sci->stereo_bands; i < sci->total_bands; i++)
434     {
435       sci->bitalloc[2 * i + 1] = 0;
436     }
437   }
```

```
438
439 static int L12_dequantize_granule(float *grbuf, bs_t *bs,
      L12_scale_info *sci, int group_size)
440 {
441   int i, j, k, choff = 576;
442   for (j = 0; j < 4; j++)
443   {
444     float *dst = grbuf + group_size * j;
445     for (i = 0; i < 2 * sci->total_bands; i++)
446     {
447       int ba = sci->bitalloc[i];
448       if (ba != 0)
449       {
450         if (ba < 17)
451         {
452           int half = (1 << (ba - 1)) - 1;
453           for (k = 0; k < group_size; k++)
454           {
455             dst[k] = (float)((int)get_bits(bs, ba) - half);
456           }
457         }
458         else
459         {
460           unsigned mod = (2 << (ba - 17)) + 1;               /* 3, 5,
                9 */
461           unsigned code = get_bits(bs, mod + 2 - (mod >> 3)); /* 5, 7,
                10 */
462           for (k = 0; k < group_size; k++, code /= mod)
463           {
464             dst[k] = (float)((int)(code % mod - mod / 2));
465           }
466         }
467       }
468       dst += choff;
469       choff = 18 - choff;
470     }
471   }
472   return group_size * 4;
473 }
474
475 static void L12_apply_scf_384(L12_scale_info *sci, const float *scf,
      float *dst)
476 {
477   int i, k;
478   memcpy(dst + 576 + sci->stereo_bands * 18, dst + sci->stereo_bands *
        18, (sci->total_bands - sci->stereo_bands) * 18 * sizeof(float));
479   for (i = 0; i < sci->total_bands; i++, dst += 18, scf += 6)
480   {
481     for (k = 0; k < 12; k++)
482     {
483       dst[k + 0] *= scf[0];
484       dst[k + 576] *= scf[3];
485     }
```

```
486     }
487   }
488   #endif /* MINIMP3_ONLY_MP3 */
489
490   static int L3_read_side_info(bs_t *bs, L3_gr_info_t *gr, const uint8_t
        *hdr)
491   {
492     static const uint8_t g_scf_long[8][23] = {
493         {6, 6, 6, 6, 6, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 38, 46,
            52, 60, 68, 58, 54, 0},
494         {12, 12, 12, 12, 12, 12, 16, 20, 24, 28, 32, 40, 48, 56, 64, 76,
             90, 2, 2, 2, 2, 2, 0},
495         {6, 6, 6, 6, 6, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 38, 46,
            52, 60, 68, 58, 54, 0},
496         {6, 6, 6, 6, 6, 6, 8, 10, 12, 14, 16, 18, 22, 26, 32, 38, 46,
            54, 62, 70, 76, 36, 0},
497         {6, 6, 6, 6, 6, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 38, 46,
            52, 60, 68, 58, 54, 0},
498         {4, 4, 4, 4, 4, 4, 6, 6, 8, 8, 10, 12, 16, 20, 24, 28, 34, 42,
            50, 54, 76, 158, 0},
499         {4, 4, 4, 4, 4, 4, 6, 6, 6, 8, 10, 12, 16, 18, 22, 28, 34, 40,
            46, 54, 54, 192, 0},
500         {4, 4, 4, 4, 4, 4, 6, 6, 8, 10, 12, 16, 20, 24, 30, 38, 46, 56,
            68, 84, 102, 26, 0}};
501     static const uint8_t g_scf_short[8][40] = {
502         {4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 6, 8, 8, 8, 10, 10, 10, 12,
            12, 12, 14, 14, 14, 18, 18, 18, 24, 24, 24, 30, 30, 30, 40, 40,
             40, 18, 18, 18, 0},
503         {8, 8, 8, 8, 8, 8, 8, 8, 8, 12, 12, 12, 16, 16, 16, 20, 20, 20,
            24, 24, 24, 28, 28, 28, 36, 36, 36, 2, 2, 2, 2, 2, 2, 2, 2, 2,
            26, 26, 26, 0},
504         {4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 8, 8, 8, 10, 10,
            10, 14, 14, 14, 18, 18, 18, 26, 26, 26, 32, 32, 32, 42, 42, 42,
             18, 18, 18, 0},
505         {4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 6, 8, 8, 8, 10, 10, 10, 12,
            12, 12, 14, 14, 14, 18, 18, 18, 24, 24, 24, 32, 32, 32, 44, 44,
             44, 12, 12, 12, 0},
506         {4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 6, 8, 8, 8, 10, 10, 10, 12,
            12, 12, 14, 14, 14, 18, 18, 18, 24, 24, 24, 30, 30, 30, 40, 40,
             40, 18, 18, 18, 0},
507         {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 6, 8, 8, 8, 10, 10,
            10, 12, 12, 12, 14, 14, 14, 18, 18, 18, 22, 22, 22, 30, 30, 30,
             56, 56, 56, 0},
508         {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 10, 10,
            10, 12, 12, 12, 14, 14, 14, 16, 16, 16, 20, 20, 20, 26, 26, 26,
             66, 66, 66, 0},
509         {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 6, 8, 8, 8, 12, 12,
            12, 16, 16, 16, 20, 20, 20, 26, 26, 26, 34, 34, 34, 42, 42, 42,
             12, 12, 12, 0}};
510     static const uint8_t g_scf_mixed[8][40] = {
511         {6, 6, 6, 6, 6, 6, 6, 6, 6, 8, 8, 8, 10, 10, 10, 12, 12, 12, 14,
             14, 14, 18, 18, 18, 24, 24, 24, 30, 30, 30, 40, 40, 40, 18,
            18, 18, 0},
```

```
512         {12, 12, 12, 4, 4, 4, 8, 8, 8, 12, 12, 12, 16, 16, 16, 20, 20,
              20, 24, 24, 24, 28, 28, 28, 36, 36, 36, 2, 2, 2, 2, 2, 2, 2, 2,
               2, 26, 26, 26, 0},
513         {6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 8, 8, 8, 10, 10, 10, 14,
              14, 14, 18, 18, 18, 26, 26, 26, 32, 32, 32, 42, 42, 42, 18, 18,
               18, 0},
514         {6, 6, 6, 6, 6, 6, 6, 6, 6, 8, 8, 8, 10, 10, 10, 12, 12, 12, 14,
               14, 14, 18, 18, 18, 24, 24, 24, 32, 32, 32, 44, 44, 44, 12,
              12, 12, 0},
515         {6, 6, 6, 6, 6, 6, 6, 6, 6, 8, 8, 8, 10, 10, 10, 12, 12, 12, 14,
               14, 14, 18, 18, 18, 24, 24, 24, 30, 30, 30, 40, 40, 40, 18,
              18, 18, 0},
516         {4, 4, 4, 4, 4, 4, 6, 6, 4, 4, 4, 6, 6, 6, 8, 8, 8, 10, 10, 10,
              12, 12, 12, 14, 14, 14, 18, 18, 18, 22, 22, 22, 30, 30, 30, 56,
               56, 56, 0},
517         {4, 4, 4, 4, 4, 4, 6, 6, 4, 4, 4, 6, 6, 6, 6, 6, 6, 10, 10, 10,
              12, 12, 12, 14, 14, 14, 16, 16, 16, 20, 20, 20, 26, 26, 26, 66,
               66, 66, 0},
518         {4, 4, 4, 4, 4, 4, 6, 6, 4, 4, 4, 6, 6, 6, 8, 8, 8, 12, 12, 12,
              16, 16, 16, 20, 20, 20, 26, 26, 26, 34, 34, 34, 42, 42, 42, 12,
               12, 12, 0}};
519
520     unsigned tables, scfsi = 0;
521     int main_data_begin, part_23_sum = 0;
522     int sr_idx = HDR_GET_MY_SAMPLE_RATE(hdr);
523     sr_idx -= (sr_idx != 0);
524     int gr_count = HDR_IS_MONO(hdr) ? 1 : 2;
525
526     if (HDR_TEST_MPEG1(hdr))
527     {
528       gr_count *= 2;
529       main_data_begin = get_bits(bs, 9);
530       scfsi = get_bits(bs, 7 + gr_count);
531     }
532     else
533     {
534       main_data_begin = get_bits(bs, 8 + gr_count) >> gr_count;
535     }
536
537     do
538     {
539       if (HDR_IS_MONO(hdr))
540       {
541         scfsi <<= 4;
542       }
543       gr->part_23_length = (uint16_t)get_bits(bs, 12);
544       part_23_sum += gr->part_23_length;
545       gr->big_values = (uint16_t)get_bits(bs, 9);
546       if (gr->big_values > 288)
547       {
548         return -1;
549       }
550       gr->global_gain = (uint8_t)get_bits(bs, 8);
```

```
551       gr->scalefac_compress = (uint16_t)get_bits(bs, HDR_TEST_MPEG1
            (hdr) ? 4 : 9);
552       gr->sfbtab = g_scf_long[sr_idx];
553       gr->n_long_sfb = 22;
554       gr->n_short_sfb = 0;
555       if (get_bits(bs, 1))
556       {
557         gr->block_type = (uint8_t)get_bits(bs, 2);
558         if (!gr->block_type)
559         {
560           return -1;
561         }
562         gr->mixed_block_flag = (uint8_t)get_bits(bs, 1);
563         gr->region_count[0] = 7;
564         gr->region_count[1] = 255;
565         if (gr->block_type == SHORT_BLOCK_TYPE)
566         {
567           scfsi &= 0x0F0F;
568           if (!gr->mixed_block_flag)
569           {
570             gr->region_count[0] = 8;
571             gr->sfbtab = g_scf_short[sr_idx];
572             gr->n_long_sfb = 0;
573             gr->n_short_sfb = 39;
574           }
575           else
576           {
577             gr->sfbtab = g_scf_mixed[sr_idx];
578             gr->n_long_sfb = HDR_TEST_MPEG1(hdr) ? 8 : 6;
579             gr->n_short_sfb = 30;
580           }
581         }
582         tables = get_bits(bs, 10);
583         tables <<= 5;
584         gr->subblock_gain[0] = (uint8_t)get_bits(bs, 3);
585         gr->subblock_gain[1] = (uint8_t)get_bits(bs, 3);
586         gr->subblock_gain[2] = (uint8_t)get_bits(bs, 3);
587       }
588       else
589       {
590         gr->block_type = 0;
591         gr->mixed_block_flag = 0;
592         tables = get_bits(bs, 15);
593         gr->region_count[0] = (uint8_t)get_bits(bs, 4);
594         gr->region_count[1] = (uint8_t)get_bits(bs, 3);
595         gr->region_count[2] = 255;
596       }
597       gr->table_select[0] = (uint8_t)(tables >> 10);
598       gr->table_select[1] = (uint8_t)((tables >> 5) & 31);
599       gr->table_select[2] = (uint8_t)((tables)&31);
600       gr->preflag = HDR_TEST_MPEG1(hdr) ? get_bits(bs, 1) : (gr-
            >scalefac_compress >= 500);
601       gr->scalefac_scale = (uint8_t)get_bits(bs, 1);
```

```
602       gr->count1_table = (uint8_t)get_bits(bs, 1);
603       gr->scfsi = (uint8_t)((scfsi >> 12) & 15);
604       scfsi <<= 4;
605       gr++;
606     } while (--gr_count);
607
608     if (part_23_sum + bs->pos > bs->limit + main_data_begin * 8)
609     {
610       return -1;
611     }
612
613     return main_data_begin;
614   }
615
616   static void L3_read_scalefactors(uint8_t *scf, uint8_t *ist_pos, const ↵
        uint8_t *scf_size, const uint8_t *scf_count, bs_t *bitbuf, int      ↵
      scfsi)
617   {
618     int i, k;
619     for (i = 0; i < 4 && scf_count[i]; i++, scfsi *= 2)
620     {
621       int cnt = scf_count[i];
622       if (scfsi & 8)
623       {
624         memcpy(scf, ist_pos, cnt);
625       }
626       else
627       {
628         int bits = scf_size[i];
629         if (!bits)
630         {
631           memset(scf, 0, cnt);
632           memset(ist_pos, 0, cnt);
633         }
634         else
635         {
636           int max_scf = (scfsi < 0) ? (1 << bits) - 1 : -1;
637           for (k = 0; k < cnt; k++)
638           {
639             int s = get_bits(bitbuf, bits);
640             ist_pos[k] = (s == max_scf ? -1 : s);
641             scf[k] = s;
642           }
643         }
644       }
645       ist_pos += cnt;
646       scf += cnt;
647     }
648     scf[0] = scf[1] = scf[2] = 0;
649   }
650
651   static float L3_ldexp_q2(float y, int exp_q2)
652   {
```

```
653     static const float g_expfrac[4] = {9.31322575e-10f, 7.83145814e-10f,
          6.58544508e-10f, 5.53767716e-10f};
654     int e;
655     do
656     {
657       e = MINIMP3_MIN(30 * 4, exp_q2);
658       y *= g_expfrac[e & 3] * (1 << 30 >> (e >> 2));
659     } while ((exp_q2 -= e) > 0);
660     return y;
661   }
662
663   static void L3_decode_scalefactors(const uint8_t *hdr, uint8_t
        *ist_pos, bs_t *bs, const L3_gr_info_t *gr, float *scf, int ch)
664   {
665     static const uint8_t g_scf_partitions[3][28] = {
666         {6, 5, 5, 5, 6, 5, 5, 5, 6, 5, 7, 3, 11, 10, 0, 0, 7, 7, 7, 0,
            6, 6, 6, 3, 8, 8, 5, 0},
667         {8, 9, 6, 12, 6, 9, 9, 9, 6, 9, 12, 6, 15, 18, 0, 0, 6, 15, 12,
            0, 6, 12, 9, 6, 6, 18, 9, 0},
668         {9, 9, 6, 12, 9, 9, 9, 9, 9, 9, 12, 6, 18, 18, 0, 0, 12, 12, 12,
             0, 12, 9, 9, 6, 15, 12, 9, 0}};
669     const uint8_t *scf_partition = g_scf_partitions[!!gr->n_short_sfb
        + !gr->n_long_sfb];
670     uint8_t scf_size[4], iscf[40];
671     int i, scf_shift = gr->scalefac_scale + 1, gain_exp, scfsi = gr-
        >scfsi;
672     float gain;
673
674     if (HDR_TEST_MPEG1(hdr))
675     {
676       static const uint8_t g_scfc_decode[16] = {0, 1, 2, 3, 12, 5, 6, 7,
            9, 10, 11, 13, 14, 15, 18, 19};
677       int part = g_scfc_decode[gr->scalefac_compress];
678       scf_size[1] = scf_size[0] = (uint8_t)(part >> 2);
679       scf_size[3] = scf_size[2] = (uint8_t)(part & 3);
680     }
681     else
682     {
683       static const uint8_t g_mod[6 * 4] = {5, 5, 4, 4, 5, 5, 4, 1, 4, 3,
            1, 1, 5, 6, 6, 1, 4, 4, 4, 1, 4, 3, 1, 1};
684       int k, modprod, sfc, ist = HDR_TEST_I_STEREO(hdr) && ch;
685       sfc = gr->scalefac_compress >> ist;
686       for (k = ist * 3 * 4; sfc >= 0; sfc -= modprod, k += 4)
687       {
688         for (modprod = 1, i = 3; i >= 0; i--)
689         {
690           scf_size[i] = (uint8_t)(sfc / modprod % g_mod[k + i]);
691           modprod *= g_mod[k + i];
692         }
693       }
694       scf_partition += k;
695       scfsi = -16;
696     }
```

```
697     L3_read_scalefactors(iscf, ist_pos, scf_size, scf_partition, bs,
          scfsi);
698
699     if (gr->n_short_sfb)
700     {
701       int sh = 3 - scf_shift;
702       for (i = 0; i < gr->n_short_sfb; i += 3)
703       {
704         iscf[gr->n_long_sfb + i + 0] += gr->subblock_gain[0] << sh;
705         iscf[gr->n_long_sfb + i + 1] += gr->subblock_gain[1] << sh;
706         iscf[gr->n_long_sfb + i + 2] += gr->subblock_gain[2] << sh;
707       }
708     }
709     else if (gr->preflag)
710     {
711       static const uint8_t g_preamp[10] = {1, 1, 1, 1, 2, 2, 3, 3, 3,
            2};
712       for (i = 0; i < 10; i++)
713       {
714         iscf[11 + i] += g_preamp[i];
715       }
716     }
717
718     gain_exp = gr->global_gain + BITS_DEQUANTIZER_OUT * 4 - 210 -
          (HDR_IS_MS_STEREO(hdr) ? 2 : 0);
719     gain = L3_ldexp_q2(1 << (MAX_SCFI / 4), MAX_SCFI - gain_exp);
720     for (i = 0; i < (int)(gr->n_long_sfb + gr->n_short_sfb); i++)
721     {
722       scf[i] = L3_ldexp_q2(gain, iscf[i] << scf_shift);
723     }
724 }
725
726 static const float g_pow43[129 + 16] = {
727       0, -1, -2.519842f, -4.326749f, -6.349604f, -8.549880f,
            -10.902724f, -13.390518f, -16.000000f, -18.720754f, -21.544347f,
            -24.463781f, -27.473142f, -30.567351f, -33.741992f, -36.993181f,
728       0, 1, 2.519842f, 4.326749f, 6.349604f, 8.549880f, 10.902724f,
            13.390518f, 16.000000f, 18.720754f, 21.544347f, 24.463781f,
            27.473142f, 30.567351f, 33.741992f, 36.993181f, 40.317474f,
            43.711787f, 47.173345f, 50.699631f, 54.288352f, 57.937408f,
            61.644865f, 65.408941f, 69.227979f, 73.100443f, 77.024898f,
            81.000000f, 85.024491f, 89.097188f, 93.216975f, 97.382800f,
            101.593667f, 105.848633f, 110.146801f, 114.487321f, 118.869381f,
            123.292209f, 127.755065f, 132.257246f, 136.798076f, 141.376907f,
            145.993119f, 150.646117f, 155.335327f, 160.060199f, 164.820202f,
            169.614826f, 174.443577f, 179.305980f, 184.201575f, 189.129918f,
            194.090580f, 199.083145f, 204.107210f, 209.162385f, 214.248292f,
            219.364564f, 224.510845f, 229.686789f, 234.892058f, 240.126328f,
            245.389280f, 250.680604f, 256.000000f, 261.347174f, 266.721841f,
            272.123723f, 277.552547f, 283.008049f, 288.489971f, 293.998060f,
            299.532071f, 305.091761f, 310.676898f, 316.287249f, 321.922592f,
            327.582707f, 333.267377f, 338.976394f, 344.709550f, 350.466646f,
            356.247482f, 362.051866f, 367.879608f, 373.730522f, 379.604427f,
```

```
          385.501143f, 391.420496f, 397.362314f, 403.326427f, 409.312672f,
          415.320884f, 421.350905f, 427.402579f, 433.475750f, 439.570269f,
          445.685987f, 451.822757f, 457.980436f, 464.158883f, 470.357960f,
          476.577530f, 482.817459f, 489.077615f, 495.357868f, 501.658090f,
          507.978156f, 514.317941f, 520.677324f, 527.056184f, 533.454404f,
          539.871867f, 546.308458f, 552.764065f, 559.238575f, 565.731879f,
          572.243870f, 578.774440f, 585.323483f, 591.890898f, 598.476581f,
          605.080431f, 611.702349f, 618.342238f, 625.000000f, 631.675540f,
          638.368763f, 645.079578f};
729
730 static float L3_pow_43(int x)
731 {
732   float frac;
733   int sign, mult = 256;
734
735   if (x < 129)
736   {
737     return g_pow43[16 + x];
738   }
739
740   if (x < 1024)
741   {
742     mult = 16;
743     x <<= 3;
744   }
745
746   sign = 2 * x & 64;
747   frac = (float)((x & 63) - sign) / ((x & ~63) + sign);
748   return g_pow43[16 + ((x + sign) >> 6)] * (1.f + frac * ((4.f / 3) +
        frac * (2.f / 9))) * mult;
749 }
750
751 static void L3_huffman(float *dst, bs_t *bs, const L3_gr_info_t
      *gr_info, const float *scf, int layer3gr_limit)
752 {
753   static const int16_t tabs[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
754                                  785, 785, 785, 785, 784, 784, 784,
                   784, 513, 513, 513, 513, 513, 513, 513, 513, 256,
                   256, 256, 256, 256, 256, 256, 256, 256, 256, 256,
                   256, 256, 256, 256, 256,
755                                  -255, 1313, 1298, 1282, 785, 785,
                   785, 785, 784, 784, 784, 784, 769, 769, 769, 769,
                   256, 256, 256, 256, 256, 256, 256, 256, 256, 256,
                   256, 256, 256, 256, 256, 256, 290, 288,
756                                  -255, 1313, 1298, 1282, 769, 769,
                   769, 769, 529, 529, 529, 529, 529, 529, 529, 529,
                   528, 528, 528, 528, 528, 528, 528, 528, 512, 512,
                   512, 512, 512, 512, 512, 512, 290, 288,
757                                  -253, -318, -351, -367, 785, 785,
                   785, 785, 784, 784, 784, 784, 769, 769, 769, 769,
                   256, 256, 256, 256, 256, 256, 256, 256, 256, 256,
                   256, 256, 256, 256, 256, 256, 819, 818, 547, 547,
```

```
                    275, 275, 275, 275, 561, 560, 515, 546, 289, 274,
                    288, 258,
758                                 -254, -287, 1329, 1299, 1314, 1312,
                    1057, 1057, 1042, 1042, 1026, 1026, 784, 784, 784,
                    784, 529, 529, 529, 529, 529, 529, 529, 529, 769,
                    769, 769, 769, 768, 768, 768, 768, 563, 560, 306,
                    306, 291, 259,
759                                 -252, -413, -477, -542, 1298, -575,
                    1041, 1041, 784, 784, 784, 784, 769, 769, 769, 769,
                    256, 256, 256, 256, 256, 256, 256, 256, 256, 256,
                    256, 256, 256, 256, 256, 256, -383, -399, 1107, 1092,
                     1106, 1061, 849, 849, 789, 789, 1104, 1091, 773,
                    773, 1076, 1075, 341, 340, 325, 309, 834, 804, 577,
                    577, 532, 532, 516, 516, 832, 818, 803, 816, 561,
                    561, 531, 531, 515, 546, 289, 289, 288, 258,
760                                 -252, -429, -493, -559, 1057, 1057,
                    1042, 1042, 529, 529, 529, 529, 529, 529, 529, 529,
                    784, 784, 784, 784, 769, 769, 769, 769, 512, 512,
                    512, 512, 512, 512, 512, 512, -382, 1077, -415, 1106,
                     1061, 1104, 849, 849, 789, 789, 1091, 1076, 1029,
                    1075, 834, 834, 597, 581, 340, 340, 339, 324, 804,
                    833, 532, 532, 832, 772, 818, 803, 817, 787, 816,
                    771, 290, 290, 290, 290, 288, 258,
761                                 -253, -349, -414, -447, -463, 1329,
                    1299, -479, 1314, 1312, 1057, 1057, 1042, 1042, 1026,
                     1026, 785, 785, 785, 785, 784, 784, 784, 784, 769,
                    769, 769, 769, 768, 768, 768, 768, -319, 851, 821,
                    -335, 836, 850, 805, 849, 341, 340, 325, 336, 533,
                    533, 579, 579, 564, 564, 773, 832, 578, 548, 563,
                    516, 321, 276, 306, 291, 304, 259,
762                                 -251, -572, -733, -830, -863, -879,
                    1041, 1041, 784, 784, 784, 784, 769, 769, 769, 769,
                    256, 256, 256, 256, 256, 256, 256, 256, 256, 256,
                    256, 256, 256, 256, 256, 256, -511, -527, -543, 1396,
                     1351, 1381, 1366, 1395, 1335, 1380, -559, 1334,
                    1138, 1138, 1063, 1063, 1350, 1392, 1031, 1031, 1062,
                     1062, 1364, 1363, 1120, 1120, 1333, 1348, 881, 881,
                    881, 881, 375, 374, 359, 373, 343, 358, 341, 325,
                    791, 791, 1123, 1122, -703, 1105, 1045, -719, 865,
                    865, 790, 790, 774, 774, 1104, 1029, 338, 293, 323,
                    308, -799, -815, 833, 788, 772, 818, 803, 816, 322,
                    292, 307, 320, 561, 531, 515, 546, 289, 274, 288,
                    258,
763                                 -251, -525, -605, -685, -765, -831,
                    -846, 1298, 1057, 1057, 1312, 1282, 785, 785, 785,
                    785, 784, 784, 784, 784, 769, 769, 769, 769, 512,
                    512, 512, 512, 512, 512, 512, 512, 1399, 1398, 1383,
                    1367, 1382, 1396, 1351, -511, 1381, 1366, 1139, 1139,
                     1079, 1079, 1124, 1124, 1364, 1349, 1363, 1333, 882,
                     882, 882, 882, 807, 807, 807, 807, 1094, 1094, 1136,
                     1136, 373, 341, 535, 535, 881, 775, 867, 822, 774,
                    -591, 324, 338, -671, 849, 550, 550, 866, 864, 609,
                    609, 293, 336, 534, 534, 789, 835, 773, -751, 834,
```

```
                804, 308, 307, 833, 788, 832, 772, 562, 562, 547,
                547, 305, 275, 560, 515, 290, 290,
764                             -252, -397, -477, -557, -622, -653,
                -719, -735, -750, 1329, 1299, 1314, 1057, 1057, 1042,
                 1042, 1312, 1282, 1024, 1024, 785, 785, 785, 785,
                784, 784, 784, 784, 769, 769, 769, 769, -383, 1127,
                1141, 1111, 1126, 1140, 1095, 1110, 869, 869, 883,
                883, 1079, 1109, 882, 882, 375, 374, 807, 868, 838,
                881, 791, -463, 867, 822, 368, 263, 852, 837, 836,
                -543, 610, 610, 550, 550, 352, 336, 534, 534, 865,
                774, 851, 821, 850, 805, 593, 533, 579, 564, 773,
                832, 578, 578, 548, 548, 577, 577, 307, 276, 306,
                291, 516, 560, 259, 259,
765                             -250, -2107, -2507, -2764, -2909,
                -2974, -3007, -3023, 1041, 1041, 1040, 1040, 769,
                769, 769, 769, 256, 256, 256, 256, 256, 256, 256,
                256, 256, 256, 256, 256, 256, 256, 256, 256, -767,
                -1052, -1213, -1277, -1358, -1405, -1469, -1535,
                -1550, -1582, -1614, -1647, -1662, -1694, -1726,
                -1759, -1774, -1807, -1822, -1854, -1886, 1565,
                -1919, -1935, -1951, -1967, 1731, 1730, 1580, 1717,
                -1983, 1729, 1564, -1999, 1548, -2015, -2031, 1715,
                1595, -2047, 1714, -2063, 1610, -2079, 1609, -2095,
                1323, 1323, 1457, 1457, 1307, 1307, 1712, 1547, 1641,
                 1700, 1699, 1594, 1685, 1625, 1442, 1442, 1322,
                1322, -780, -973, -910, 1279, 1278, 1277, 1262, 1276,
                 1261, 1275, 1215, 1260, 1229, -959, 974, 974, 989,
                989, -943, 735, 478, 478, 495, 463, 506, 414, -1039,
                1003, 958, 1017, 927, 942, 987, 957, 431, 476, 1272,
                1167, 1228, -1183, 1256, -1199, 895, 895, 941, 941,
                1242, 1227, 1212, 1135, 1014, 1014, 490, 489, 503,
                487, 910, 1013, 985, 925, 863, 894, 970, 955, 1012,
                847, -1343, 831, 755, 755, 984, 909, 428, 366, 754,
                559, -1391, 752, 486, 457, 924, 997, 698, 698, 983,
                893, 740, 740, 908, 877, 739, 739, 667, 667, 953,
                938, 497, 287, 271, 271, 683, 606, 590, 712, 726,
                574, 302, 302, 738, 736, 481, 286, 526, 725, 605,
                711, 636, 724, 696, 651, 589, 681, 666, 710, 364,
                467, 573, 695, 466, 466, 301, 465, 379, 379, 709,
                604, 665, 679, 316, 316, 634, 633, 436, 436, 464,
                269, 424, 394, 452, 332, 438, 363, 347, 408, 393,
                448, 331, 422, 362, 407, 392, 421, 346, 406, 391,
                376, 375, 359, 1441, 1306, -2367, 1290, -2383, 1337,
                -2399, -2415, 1426, 1321, -2431, 1411, 1336, -2447,
                -2463, -2479, 1169, 1169, 1049, 1049, 1424, 1289,
                1412, 1352, 1319, -2495, 1154, 1154, 1064, 1064,
                1153, 1153, 416, 390, 360, 404, 403, 389, 344, 374,
                373, 343, 358, 372, 327, 357, 342, 311, 356, 326,
                1395, 1394, 1137, 1137, 1047, 1047, 1365, 1392, 1287,
                 1379, 1334, 1364, 1349, 1378, 1318, 1363, 792, 792,
                792, 792, 1152, 1152, 1032, 1032, 1121, 1121, 1046,
                1046, 1120, 1120, 1030, 1030, -2895, 1106, 1061,
                1104, 849, 849, 789, 789, 1091, 1076, 1029, 1090,
```

```
                    1060, 1075, 833, 833, 309, 324, 532, 532, 832, 772,
                    818, 803, 561, 561, 531, 560, 515, 546, 289, 274,
                    288, 258,
766                                 -250, -1179, -1579, -1836, -1996,
                    -2124, -2253, -2333, -2413, -2477, -2542, -2574,
                    -2607, -2622, -2655, 1314, 1313, 1298, 1312, 1282,
                    785, 785, 785, 785, 1040, 1040, 1025, 1025, 768, 768,
                     768, 768, -766, -798, -830, -862, -895, -911, -927,
                    -943, -959, -975, -991, -1007, -1023, -1039, -1055,
                    -1070, 1724, 1647, -1103, -1119, 1631, 1767, 1662,
                    1738, 1708, 1723, -1135, 1780, 1615, 1779, 1599,
                    1677, 1646, 1778, 1583, -1151, 1777, 1567, 1737,
                    1692, 1765, 1722, 1707, 1630, 1751, 1661, 1764, 1614,
                     1736, 1676, 1763, 1750, 1645, 1598, 1721, 1691,
                    1762, 1706, 1582, 1761, 1566, -1167, 1749, 1629, 767,
                     766, 751, 765, 494, 494, 735, 764, 719, 749, 734,
                    763, 447, 447, 748, 718, 477, 506, 431, 491, 446,
                    476, 461, 505, 415, 430, 475, 445, 504, 399, 460,
                    489, 414, 503, 383, 474, 429, 459, 502, 502, 746,
                    752, 488, 398, 501, 473, 413, 472, 486, 271, 480,
                    270, -1439, -1455, 1357, -1471, -1487, -1503, 1341,
                    1325, -1519, 1489, 1463, 1403, 1309, -1535, 1372,
                    1448, 1418, 1476, 1356, 1462, 1387, -1551, 1475,
                    1340, 1447, 1402, 1386, -1567, 1068, 1068, 1474,
                    1461, 455, 380, 468, 440, 395, 425, 410, 454, 364,
                    467, 466, 464, 453, 269, 409, 448, 268, 432, 1371,
                    1473, 1432, 1417, 1308, 1460, 1355, 1446, 1459, 1431,
                     1083, 1083, 1401, 1416, 1458, 1445, 1067, 1067,
                    1370, 1457, 1051, 1051, 1291, 1430, 1385, 1444, 1354,
                     1415, 1400, 1443, 1082, 1082, 1173, 1113, 1186,
                    1066, 1185, 1050, -1967, 1158, 1128, 1172, 1097,
                    1171, 1081, -1983, 1157, 1112, 416, 266, 375, 400,
                    1170, 1142, 1127, 1065, 793, 793, 1169, 1033, 1156,
                    1096, 1141, 1111, 1155, 1080, 1126, 1140, 898, 898,
                    808, 808, 897, 897, 792, 792, 1095, 1152, 1032, 1125,
                     1110, 1139, 1079, 1124, 882, 807, 838, 881, 853,
                    791, -2319, 867, 368, 263, 822, 852, 837, 866, 806,
                    865, -2399, 851, 352, 262, 534, 534, 821, 836, 594,
                    594, 549, 549, 593, 593, 533, 533, 848, 773, 579,
                    579, 564, 578, 548, 563, 276, 276, 577, 576, 306,
                    291, 516, 560, 305, 305, 275, 259,
767                                 -251, -892, -2058, -2620, -2828,
                    -2957, -3023, -3039, 1041, 1041, 1040, 1040, 769,
                    769, 769, 769, 256, 256, 256, 256, 256, 256, 256,
                    256, 256, 256, 256, 256, 256, 256, 256, 256, -511,
                    -527, -543, -559, 1530, -575, -591, 1528, 1527, 1407,
                     1526, 1391, 1023, 1023, 1023, 1023, 1525, 1375,
                    1268, 1268, 1103, 1103, 1087, 1087, 1039, 1039, 1523,
                     -604, 815, 815, 815, 815, 510, 495, 509, 479, 508,
                    463, 507, 447, 431, 505, 415, 399, -734, -782, 1262,
                    -815, 1259, 1244, -831, 1258, 1228, -847, -863, 1196,
                     -879, 1253, 987, 987, 748, -767, 493, 493, 462, 477,
                     414, 414, 686, 669, 478, 446, 461, 445, 474, 429,
```

```
                487, 458, 412, 471, 1266, 1264, 1009, 1009, 799, 799,
                 -1019, -1276, -1452, -1581, -1677, -1757, -1821,
                -1886, -1933, -1997, 1257, 1257, 1483, 1468, 1512,
                1422, 1497, 1406, 1467, 1496, 1421, 1510, 1134, 1134,
                 1225, 1225, 1466, 1451, 1374, 1405, 1252, 1252,
                1358, 1480, 1164, 1164, 1251, 1251, 1238, 1238, 1389,
                 1465, -1407, 1054, 1101, -1423, 1207, -1439, 830,
                830, 1248, 1038, 1237, 1117, 1223, 1148, 1236, 1208,
                411, 426, 395, 410, 379, 269, 1193, 1222, 1132, 1235,
                 1221, 1116, 976, 976, 1192, 1162, 1177, 1220, 1131,
                1191, 963, 963, -1647, 961, 780, -1663, 558, 558,
                994, 993, 437, 408, 393, 407, 829, 978, 813, 797,
                947, -1743, 721, 721, 377, 392, 844, 950, 828, 890,
                706, 706, 812, 859, 796, 960, 948, 843, 934, 874,
                571, 571, -1919, 690, 555, 689, 421, 346, 539, 539,
                944, 779, 918, 873, 932, 842, 903, 888, 570, 570,
                931, 917, 674, 674, -2575, 1562, -2591, 1609, -2607,
                1654, 1322, 1322, 1441, 1441, 1696, 1546, 1683, 1593,
                 1669, 1624, 1426, 1426, 1321, 1321, 1639, 1680,
                1425, 1425, 1305, 1305, 1545, 1668, 1608, 1623, 1667,
                 1592, 1638, 1666, 1320, 1320, 1652, 1607, 1409,
                1409, 1304, 1304, 1288, 1288, 1664, 1637, 1395, 1395,
                 1335, 1335, 1622, 1636, 1394, 1394, 1319, 1319,
                1606, 1621, 1392, 1392, 1137, 1137, 1137, 1137, 345,
                390, 360, 375, 404, 373, 1047, -2751, -2767, -2783,
                1062, 1121, 1046, -2799, 1077, -2815, 1106, 1061,
                789, 789, 1105, 1104, 263, 355, 310, 340, 325, 354,
                352, 262, 339, 324, 1091, 1076, 1029, 1090, 1060,
                1075, 833, 833, 788, 788, 1088, 1028, 818, 818, 803,
                803, 561, 561, 531, 531, 816, 771, 546, 546, 289,
                274, 288, 258,
768                                 -253, -317, -381, -446, -478, -509,
                1279, 1279, -811, -1179, -1451, -1756, -1900, -2028,
                -2189, -2253, -2333, -2414, -2445, -2511, -2526,
                1313, 1298, -2559, 1041, 1041, 1040, 1040, 1025,
                1025, 1024, 1024, 1022, 1007, 1021, 991, 1020, 975,
                1019, 959, 687, 687, 1018, 1017, 671, 671, 655, 655,
                1016, 1015, 639, 639, 758, 758, 623, 623, 757, 607,
                756, 591, 755, 575, 754, 559, 543, 543, 1009, 783,
                -575, -621, -685, -749, 496, -590, 750, 749, 734,
                748, 974, 989, 1003, 958, 988, 973, 1002, 942, 987,
                957, 972, 1001, 926, 986, 941, 971, 956, 1000, 910,
                985, 925, 999, 894, 970, -1071, -1087, -1102, 1390,
                -1135, 1436, 1509, 1451, 1374, -1151, 1405, 1358,
                1480, 1420, -1167, 1507, 1494, 1389, 1342, 1465,
                1435, 1450, 1326, 1505, 1310, 1493, 1373, 1479, 1404,
                 1492, 1464, 1419, 428, 443, 472, 397, 736, 526, 464,
                 464, 486, 457, 442, 471, 484, 482, 1357, 1449, 1434,
                 1478, 1388, 1491, 1341, 1490, 1325, 1489, 1463,
                1403, 1309, 1477, 1372, 1448, 1418, 1433, 1476, 1356,
                 1462, 1387, -1439, 1475, 1340, 1447, 1402, 1474,
                1324, 1461, 1371, 1473, 269, 448, 1432, 1417, 1308,
                1460, -1711, 1459, -1727, 1441, 1099, 1099, 1446,
```

```
                      1386, 1431, 1401, -1743, 1289, 1083, 1083, 1160,
                      1160, 1458, 1445, 1067, 1067, 1370, 1457, 1307, 1430,
                       1129, 1129, 1098, 1098, 268, 432, 267, 416, 266,
                      400, -1887, 1144, 1187, 1082, 1173, 1113, 1186, 1066,
                       1050, 1158, 1128, 1143, 1172, 1097, 1171, 1081, 420,
                       391, 1157, 1112, 1170, 1142, 1127, 1065, 1169, 1049,
                       1156, 1096, 1141, 1111, 1155, 1080, 1126, 1154,
                      1064, 1153, 1140, 1095, 1048, -2159, 1125, 1110,
                      1137, -2175, 823, 823, 1139, 1138, 807, 807, 384,
                      264, 368, 263, 868, 838, 853, 791, 867, 822, 852,
                      837, 866, 806, 865, 790, -2319, 851, 821, 836, 352,
                      262, 850, 805, 849, -2399, 533, 533, 835, 820, 336,
                      261, 578, 548, 563, 577, 532, 532, 832, 772, 562,
                      562, 547, 547, 305, 275, 560, 515, 290, 290, 288,
                      258};
769   static const uint8_t tab32[] = {130, 162, 193, 209, 44, 28, 76, 140,
         9, 9, 9, 9, 9, 9, 9, 9, 190, 254, 222, 238, 126, 94, 157, 157,
        109, 61, 173, 205};
770   static const uint8_t tab33[] = {252, 236, 220, 204, 188, 172, 156,
        140, 124, 108, 92, 76, 60, 44, 28, 12};
771   static const int16_t tabindex[2 * 16] = {0, 32, 64, 98, 0, 132, 180,
         218, 292, 364, 426, 538, 648, 746, 0, 1126, 1460, 1460, 1460,
        1460, 1460, 1460, 1460, 1460, 1842, 1842, 1842, 1842, 1842, 1842,
        1842, 1842};
772   static const uint8_t g_linbits[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 1, 2, 3, 4, 6, 8, 10, 13, 4, 5, 6, 7, 8, 9, 11,
        13};
773
774 #define PEEK_BITS(n) (bs_cache >> (32 - n))
775 #define FLUSH_BITS(n) \
776   {                   \
777     bs_cache <<= (n); \
778     bs_sh += (n);     \
779   }
780 #define CHECK_BITS                                  \
781   while (bs_sh >= 0)                                \
782   {                                                 \
783     bs_cache |= (uint32_t)*bs_next_ptr++ << bs_sh; \
784     bs_sh -= 8;                                     \
785   }
786 #define BSPOS ((bs_next_ptr - bs->buf) * 8 - 24 + bs_sh)
787
788   float one = 0.0f;
789   int ireg = 0, big_val_cnt = gr_info->big_values;
790   const uint8_t *sfb = gr_info->sfbtab;
791   const uint8_t *bs_next_ptr = bs->buf + bs->pos / 8;
792   uint32_t bs_cache = (((bs_next_ptr[0] * 256u + bs_next_ptr[1]) *
        256u + bs_next_ptr[2]) * 256u + bs_next_ptr[3]) << (bs->pos & 7);
793   int pairs_to_decode, np, bs_sh = (bs->pos & 7) - 8;
794   bs_next_ptr += 4;
795
796   while (big_val_cnt > 0)
797   {
```

```
798       int tab_num = gr_info->table_select[ireg];
799       int sfb_cnt = gr_info->region_count[ireg++];
800       const int16_t *codebook = tabs + tabindex[tab_num];
801       int linbits = g_linbits[tab_num];
802       if (linbits)
803       {
804         do
805         {
806           np = *sfb++ / 2;
807           pairs_to_decode = MINIMP3_MIN(big_val_cnt, np);
808           one = *scf++;
809           do
810           {
811             int j, w = 5;
812             int leaf = codebook[PEEK_BITS(w)];
813             while (leaf < 0)
814             {
815               FLUSH_BITS(w);
816               w = leaf & 7;
817               leaf = codebook[PEEK_BITS(w) - (leaf >> 3)];
818             }
819             FLUSH_BITS(leaf >> 8);
820
821             for (j = 0; j < 2; j++, dst++, leaf >>= 4)
822             {
823               int lsb = leaf & 0x0F;
824               if (lsb == 15)
825               {
826                 lsb += PEEK_BITS(linbits);
827                 FLUSH_BITS(linbits);
828                 CHECK_BITS;
829                 *dst = one * L3_pow_43(lsb) * ((int32_t)bs_cache < 0 ?
                    -1 : 1);
830               }
831               else
832               {
833                 *dst = g_pow43[16 + lsb - 16 * (bs_cache >> 31)] * one;
834               }
835               FLUSH_BITS(lsb ? 1 : 0);
836             }
837             CHECK_BITS;
838           } while (--pairs_to_decode);
839         } while ((big_val_cnt -= np) > 0 && --sfb_cnt >= 0);
840       }
841       else
842       {
843         do
844         {
845           np = *sfb++ / 2;
846           pairs_to_decode = MINIMP3_MIN(big_val_cnt, np);
847           one = *scf++;
848           do
849           {
```

```
850           int j, w = 5;
851           int leaf = codebook[PEEK_BITS(w)];
852           while (leaf < 0)
853           {
854             FLUSH_BITS(w);
855             w = leaf & 7;
856             leaf = codebook[PEEK_BITS(w) - (leaf >> 3)];
857           }
858           FLUSH_BITS(leaf >> 8);
859
860           for (j = 0; j < 2; j++, dst++, leaf >>= 4)
861           {
862             int lsb = leaf & 0x0F;
863             *dst = g_pow43[16 + lsb - 16 * (bs_cache >> 31)] * one;
864             FLUSH_BITS(lsb ? 1 : 0);
865           }
866           CHECK_BITS;
867         } while (--pairs_to_decode);
868       } while ((big_val_cnt -= np) > 0 && --sfb_cnt >= 0);
869     }
870   }
871
872   for (np = 1 - big_val_cnt;; dst += 4)
873   {
874     const uint8_t *codebook_count1 = (gr_info->count1_table) ? tab33 : ↵
        tab32;
875     int leaf = codebook_count1[PEEK_BITS(4)];
876     if (!(leaf & 8))
877     {
878       leaf = codebook_count1[(leaf >> 3) + (bs_cache << 4 >> (32 -     ↵
          (leaf & 3)))];
879     }
880     FLUSH_BITS(leaf & 7);
881     if (BSPOS > layer3gr_limit)
882     {
883       break;
884     }
885 #define RELOAD_SCALEFACTOR \
886   if (!--np)               \
887   {                        \
888     np = *sfb++ / 2;       \
889     if (!np)               \
890       break;               \
891     one = *scf++;          \
892   }
893 #define DEQ_COUNT1(s)                              \
894   if (leaf & (128 >> s))                           \
895   {                                                \
896     dst[s] = ((int32_t)bs_cache < 0) ? -one : one; \
897     FLUSH_BITS(1)                                  \
898   }
899     RELOAD_SCALEFACTOR;
900     DEQ_COUNT1(0);
```

```
901       DEQ_COUNT1(1);
902       RELOAD_SCALEFACTOR;
903       DEQ_COUNT1(2);
904       DEQ_COUNT1(3);
905       CHECK_BITS;
906     }
907
908     bs->pos = layer3gr_limit;
909   }
910
911   static void L3_midside_stereo(float *left, int n)
912   {
913     int i = 0;
914     float *right = left + 576;
915   #if HAVE_SIMD
916     if (have_simd())
917     {
918       for (; i < n - 3; i += 4)
919       {
920         f4 vl = VLD(left + i);
921         f4 vr = VLD(right + i);
922         VSTORE(left + i, VADD(vl, vr));
923         VSTORE(right + i, VSUB(vl, vr));
924       }
925   #ifdef __GNUC__
926       /* Workaround for spurious -Waggressive-loop-optimizations warning
          from gcc.
927            * For more info see: https://github.com/lieff/minimp3/
              issues/88
928            */
929       if (__builtin_constant_p(n % 4 == 0) && n % 4 == 0)
930         return;
931   #endif
932     }
933   #endif /* HAVE_SIMD */
934     for (; i < n; i++)
935     {
936       float a = left[i];
937       float b = right[i];
938       left[i] = a + b;
939       right[i] = a - b;
940     }
941   }
942
943   static void L3_intensity_stereo_band(float *left, int n, float kl,
      float kr)
944   {
945     int i;
946     for (i = 0; i < n; i++)
947     {
948       left[i + 576] = left[i] * kr;
949       left[i] = left[i] * kl;
950     }
```

```
951 }
952
953 static void L3_stereo_top_band(const float *right, const uint8_t *sfb,
       int nbands, int max_band[3])
954 {
955   int i, k;
956
957   max_band[0] = max_band[1] = max_band[2] = -1;
958
959   for (i = 0; i < nbands; i++)
960   {
961     for (k = 0; k < sfb[i]; k += 2)
962     {
963       if (right[k] != 0 || right[k + 1] != 0)
964       {
965         max_band[i % 3] = i;
966         break;
967       }
968     }
969     right += sfb[i];
970   }
971 }
972
973 static void L3_stereo_process(float *left, const uint8_t *ist_pos,
     const uint8_t *sfb, const uint8_t *hdr, int max_band[3], int
     mpeg2_sh)
974 {
975   static const float g_pan[7 * 2] = {0, 1, 0.21132487f, 0.78867513f,
       0.36602540f, 0.63397460f, 0.5f, 0.5f, 0.63397460f, 0.36602540f,
       0.78867513f, 0.21132487f, 1, 0};
976   unsigned i, max_pos = HDR_TEST_MPEG1(hdr) ? 7 : 64;
977
978   for (i = 0; sfb[i]; i++)
979   {
980     unsigned ipos = ist_pos[i];
981     if ((int)i > max_band[i % 3] && ipos < max_pos)
982     {
983       float kl, kr, s = HDR_TEST_MS_STEREO(hdr) ? 1.41421356f : 1;
984       if (HDR_TEST_MPEG1(hdr))
985       {
986         kl = g_pan[2 * ipos];
987         kr = g_pan[2 * ipos + 1];
988       }
989       else
990       {
991         kl = 1;
992         kr = L3_ldexp_q2(1, (ipos + 1) >> 1 << mpeg2_sh);
993         if (ipos & 1)
994         {
995           kl = kr;
996           kr = 1;
997         }
998       }
```

```
 999         L3_intensity_stereo_band(left, sfb[i], kl * s, kr * s);
1000       }
1001       else if (HDR_TEST_MS_STEREO(hdr))
1002       {
1003         L3_midside_stereo(left, sfb[i]);
1004       }
1005       left += sfb[i];
1006     }
1007   }
1008
1009   static void L3_intensity_stereo(float *left, uint8_t *ist_pos, const
       L3_gr_info_t *gr, const uint8_t *hdr)
1010   {
1011     int max_band[3], n_sfb = gr->n_long_sfb + gr->n_short_sfb;
1012     int i, max_blocks = gr->n_short_sfb ? 3 : 1;
1013
1014     L3_stereo_top_band(left + 576, gr->sfbtab, n_sfb, max_band);
1015     if (gr->n_long_sfb)
1016     {
1017       max_band[0] = max_band[1] = max_band[2] = MINIMP3_MAX(MINIMP3_MAX
         (max_band[0], max_band[1]), max_band[2]);
1018     }
1019     for (i = 0; i < max_blocks; i++)
1020     {
1021       int default_pos = HDR_TEST_MPEG1(hdr) ? 3 : 0;
1022       int itop = n_sfb - max_blocks + i;
1023       int prev = itop - max_blocks;
1024       ist_pos[itop] = max_band[i] >= prev ? default_pos : ist_pos[prev];
1025     }
1026     L3_stereo_process(left, ist_pos, gr->sfbtab, hdr, max_band, gr
         [1].scalefac_compress & 1);
1027   }
1028
1029   static void L3_reorder(float *grbuf, float *scratch, const uint8_t
       *sfb)
1030   {
1031     int i, len;
1032     float *src = grbuf, *dst = scratch;
1033
1034     for (; 0 != (len = *sfb); sfb += 3, src += 2 * len)
1035     {
1036       for (i = 0; i < len; i++, src++)
1037       {
1038         *dst++ = src[0 * len];
1039         *dst++ = src[1 * len];
1040         *dst++ = src[2 * len];
1041       }
1042     }
1043     memcpy(grbuf, scratch, (dst - scratch) * sizeof(float));
1044   }
1045
1046   static void L3_antialias(float *grbuf, int nbands)
1047   {
```

```
1048    static const float g_aa[2][8] = {
1049        {0.85749293f, 0.88174200f, 0.94962865f, 0.98331459f,
            0.99551782f, 0.99916056f, 0.99989920f, 0.99999316f},
1050        {0.51449576f, 0.47173197f, 0.31337745f, 0.18191320f,
            0.09457419f, 0.04096558f, 0.01419856f, 0.00369997f}};
1051
1052    for (; nbands > 0; nbands--, grbuf += 18)
1053    {
1054      int i = 0;
1055  #if HAVE_SIMD
1056      if (have_simd())
1057        for (; i < 8; i += 4)
1058        {
1059          f4 vu = VLD(grbuf + 18 + i);
1060          f4 vd = VLD(grbuf + 14 - i);
1061          f4 vc0 = VLD(g_aa[0] + i);
1062          f4 vc1 = VLD(g_aa[1] + i);
1063          vd = VREV(vd);
1064          VSTORE(grbuf + 18 + i, VSUB(VMUL(vu, vc0), VMUL(vd, vc1)));
1065          vd = VADD(VMUL(vu, vc1), VMUL(vd, vc0));
1066          VSTORE(grbuf + 14 - i, VREV(vd));
1067        }
1068  #endif /* HAVE_SIMD */
1069  #ifndef MINIMP3_ONLY_SIMD
1070      for (; i < 8; i++)
1071      {
1072        float u = grbuf[18 + i];
1073        float d = grbuf[17 - i];
1074        grbuf[18 + i] = u * g_aa[0][i] - d * g_aa[1][i];
1075        grbuf[17 - i] = u * g_aa[1][i] + d * g_aa[0][i];
1076      }
1077  #endif /* MINIMP3_ONLY_SIMD */
1078    }
1079  }
1080
1081  static void L3_dct3_9(float *y)
1082  {
1083    float s0, s1, s2, s3, s4, s5, s6, s7, s8, t0, t2, t4;
1084
1085    s0 = y[0];
1086    s2 = y[2];
1087    s4 = y[4];
1088    s6 = y[6];
1089    s8 = y[8];
1090    t0 = s0 + s6 * 0.5f;
1091    s0 -= s6;
1092    t4 = (s4 + s2) * 0.93969262f;
1093    t2 = (s8 + s2) * 0.76604444f;
1094    s6 = (s4 - s8) * 0.17364818f;
1095    s4 += s8 - s2;
1096
1097    s2 = s0 - s4 * 0.5f;
1098    y[4] = s4 + s0;
```

```
1099    s8 = t0 - t2 + s6;
1100    s0 = t0 - t4 + t2;
1101    s4 = t0 + t4 - s6;
1102
1103    s1 = y[1];
1104    s3 = y[3];
1105    s5 = y[5];
1106    s7 = y[7];
1107
1108    s3 *= 0.86602540f;
1109    t0 = (s5 + s1) * 0.98480775f;
1110    t4 = (s5 - s7) * 0.34202014f;
1111    t2 = (s1 + s7) * 0.64278761f;
1112    s1 = (s1 - s5 - s7) * 0.86602540f;
1113
1114    s5 = t0 - s3 - t2;
1115    s7 = t4 - s3 - t0;
1116    s3 = t4 + s3 - t2;
1117
1118    y[0] = s4 - s7;
1119    y[1] = s2 + s1;
1120    y[2] = s0 - s3;
1121    y[3] = s8 + s5;
1122    y[5] = s8 - s5;
1123    y[6] = s0 + s3;
1124    y[7] = s2 - s1;
1125    y[8] = s4 + s7;
1126  }
1127
1128  static void L3_imdct36(float *grbuf, float *overlap, const float       ↵
        *window, int nbands)
1129  {
1130    int i, j;
1131    static const float g_twid9[18] = {
1132        0.73727734f, 0.79335334f, 0.84339145f, 0.88701083f, 0.92387953f, ↵
             0.95371695f, 0.97629601f, 0.99144486f, 0.99904822f,           ↵
            0.67559021f, 0.60876143f, 0.53729961f, 0.46174861f,            ↵
            0.38268343f, 0.30070580f, 0.21643961f, 0.13052619f,            ↵
            0.04361938f};
1133
1134    for (j = 0; j < nbands; j++, grbuf += 18, overlap += 9)
1135    {
1136      float co[9], si[9];
1137      co[0] = -grbuf[0];
1138      si[0] = grbuf[17];
1139      for (i = 0; i < 4; i++)
1140      {
1141        si[8 - 2 * i] = grbuf[4 * i + 1] - grbuf[4 * i + 2];
1142        co[1 + 2 * i] = grbuf[4 * i + 1] + grbuf[4 * i + 2];
1143        si[7 - 2 * i] = grbuf[4 * i + 4] - grbuf[4 * i + 3];
1144        co[2 + 2 * i] = -(grbuf[4 * i + 3] + grbuf[4 * i + 4]);
1145      }
1146      L3_dct3_9(co);
```

```
1147      L3_dct3_9(si);
1148
1149      si[1] = -si[1];
1150      si[3] = -si[3];
1151      si[5] = -si[5];
1152      si[7] = -si[7];
1153
1154      i = 0;
1155
1156  #if HAVE_SIMD
1157      if (have_simd())
1158        for (; i < 8; i += 4)
1159        {
1160          f4 vovl = VLD(overlap + i);
1161          f4 vc = VLD(co + i);
1162          f4 vs = VLD(si + i);
1163          f4 vr0 = VLD(g_twid9 + i);
1164          f4 vr1 = VLD(g_twid9 + 9 + i);
1165          f4 vw0 = VLD(window + i);
1166          f4 vw1 = VLD(window + 9 + i);
1167          f4 vsum = VADD(VMUL(vc, vr1), VMUL(vs, vr0));
1168          VSTORE(overlap + i, VSUB(VMUL(vc, vr0), VMUL(vs, vr1)));
1169          VSTORE(grbuf + i, VSUB(VMUL(vovl, vw0), VMUL(vsum, vw1)));
1170          vsum = VADD(VMUL(vovl, vw1), VMUL(vsum, vw0));
1171          VSTORE(grbuf + 14 - i, VREV(vsum));
1172        }
1173  #endif /* HAVE_SIMD */
1174      for (; i < 9; i++)
1175      {
1176        float ovl = overlap[i];
1177        float sum = co[i] * g_twid9[9 + i] + si[i] * g_twid9[0 + i];
1178        overlap[i] = co[i] * g_twid9[0 + i] - si[i] * g_twid9[9 + i];
1179        grbuf[i] = ovl * window[0 + i] - sum * window[9 + i];
1180        grbuf[17 - i] = ovl * window[9 + i] + sum * window[0 + i];
1181      }
1182    }
1183  }
1184
1185  static void L3_idct3(float x0, float x1, float x2, float *dst)
1186  {
1187    float m1 = x1 * 0.86602540f;
1188    float a1 = x0 - x2 * 0.5f;
1189    dst[1] = x0 + x2;
1190    dst[0] = a1 + m1;
1191    dst[2] = a1 - m1;
1192  }
1193
1194  static void L3_imdct12(float *x, float *dst, float *overlap)
1195  {
1196    static const float g_twid3[6] = {0.79335334f, 0.92387953f,
       0.99144486f, 0.60876143f, 0.38268343f, 0.13052619f};
1197    float co[3], si[3];
1198    int i;
```

```
1199
1200    L3_idct3(-x[0], x[6] + x[3], x[12] + x[9], co);
1201    L3_idct3(x[15], x[12] - x[9], x[6] - x[3], si);
1202    si[1] = -si[1];
1203
1204    for (i = 0; i < 3; i++)
1205    {
1206      float ovl = overlap[i];
1207      float sum = co[i] * g_twid3[3 + i] + si[i] * g_twid3[0 + i];
1208      overlap[i] = co[i] * g_twid3[0 + i] - si[i] * g_twid3[3 + i];
1209      dst[i] = ovl * g_twid3[2 - i] - sum * g_twid3[5 - i];
1210      dst[5 - i] = ovl * g_twid3[5 - i] + sum * g_twid3[2 - i];
1211    }
1212  }
1213
1214  static void L3_imdct_short(float *grbuf, float *overlap, int nbands)
1215  {
1216    for (; nbands > 0; nbands--, overlap += 9, grbuf += 18)
1217    {
1218      float tmp[18];
1219      memcpy(tmp, grbuf, sizeof(tmp));
1220      memcpy(grbuf, overlap, 6 * sizeof(float));
1221      L3_imdct12(tmp, grbuf + 6, overlap + 6);
1222      L3_imdct12(tmp + 1, grbuf + 12, overlap + 6);
1223      L3_imdct12(tmp + 2, overlap, overlap + 6);
1224    }
1225  }
1226
1227  static void L3_change_sign(float *grbuf)
1228  {
1229    int b, i;
1230    for (b = 0, grbuf += 18; b < 32; b += 2, grbuf += 36)
1231      for (i = 1; i < 18; i += 2)
1232        grbuf[i] = -grbuf[i];
1233  }
1234
1235  static void L3_imdct_gr(float *grbuf, float *overlap, unsigned
       block_type, unsigned n_long_bands)
1236  {
1237    static const float g_mdct_window[2][18] = {
1238        {0.99904822f, 0.99144486f, 0.97629601f, 0.95371695f,
           0.92387953f, 0.88701083f, 0.84339145f, 0.79335334f,
           0.73727734f, 0.04361938f, 0.13052619f, 0.21643961f,
           0.30070580f, 0.38268343f, 0.46174861f, 0.53729961f,
           0.60876143f, 0.67559021f},
1239        {1, 1, 1, 1, 1, 1, 0.99144486f, 0.92387953f, 0.79335334f, 0, 0,
           0, 0, 0, 0, 0.13052619f, 0.38268343f, 0.60876143f}};
1240    if (n_long_bands)
1241    {
1242      L3_imdct36(grbuf, overlap, g_mdct_window[0], n_long_bands);
1243      grbuf += 18 * n_long_bands;
1244      overlap += 9 * n_long_bands;
1245    }
```

```
1246    if (block_type == SHORT_BLOCK_TYPE)
1247      L3_imdct_short(grbuf, overlap, 32 - n_long_bands);
1248    else
1249      L3_imdct36(grbuf, overlap, g_mdct_window[block_type ==
          STOP_BLOCK_TYPE], 32 - n_long_bands);
1250  }
1251
1252  static void L3_save_reservoir(mp3dec_t *h, mp3dec_scratch_t *s)
1253  {
1254    int pos = (s->bs.pos + 7) / 8u;
1255    int remains = s->bs.limit / 8u - pos;
1256    if (remains > MAX_BITRESERVOIR_BYTES)
1257    {
1258      pos += remains - MAX_BITRESERVOIR_BYTES;
1259      remains = MAX_BITRESERVOIR_BYTES;
1260    }
1261    if (remains > 0)
1262    {
1263      memmove(h->reserv_buf, s->maindata + pos, remains);
1264    }
1265    h->reserv = remains;
1266  }
1267
1268  static int L3_restore_reservoir(mp3dec_t *h, bs_t *bs,
       mp3dec_scratch_t *s, int main_data_begin)
1269  {
1270    int frame_bytes = (bs->limit - bs->pos) / 8;
1271    int bytes_have = MINIMP3_MIN(h->reserv, main_data_begin);
1272    memcpy(s->maindata, h->reserv_buf + MINIMP3_MAX(0, h->reserv -
         main_data_begin), MINIMP3_MIN(h->reserv, main_data_begin));
1273    memcpy(s->maindata + bytes_have, bs->buf + bs->pos / 8,
         frame_bytes);
1274    bs_init(&s->bs, s->maindata, bytes_have + frame_bytes);
1275    return h->reserv >= main_data_begin;
1276  }
1277
1278  static void L3_decode(mp3dec_t *h, mp3dec_scratch_t *s, L3_gr_info_t
       *gr_info, int nch)
1279  {
1280    int ch;
1281
1282    for (ch = 0; ch < nch; ch++)
1283    {
1284      int layer3gr_limit = s->bs.pos + gr_info[ch].part_23_length;
1285      L3_decode_scalefactors(h->header, s->ist_pos[ch], &s->bs, gr_info
           + ch, s->scf, ch);
1286      L3_huffman(s->grbuf[ch], &s->bs, gr_info + ch, s->scf,
           layer3gr_limit);
1287    }
1288
1289    if (HDR_TEST_I_STEREO(h->header))
1290    {
1291      L3_intensity_stereo(s->grbuf[0], s->ist_pos[1], gr_info, h-
```

```
        >header);
1292    }
1293    else if (HDR_IS_MS_STEREO(h->header))
1294    {
1295      L3_midside_stereo(s->grbuf[0], 576);
1296    }
1297
1298    for (ch = 0; ch < nch; ch++, gr_info++)
1299    {
1300      int aa_bands = 31;
1301      int n_long_bands = (gr_info->mixed_block_flag ? 2 : 0) << (int)
          (HDR_GET_MY_SAMPLE_RATE(h->header) == 2);
1302
1303      if (gr_info->n_short_sfb)
1304      {
1305        aa_bands = n_long_bands - 1;
1306        L3_reorder(s->grbuf[ch] + n_long_bands * 18, s->syn[0], gr_info-
            >sfbtab + gr_info->n_long_sfb);
1307      }
1308
1309      L3_antialias(s->grbuf[ch], aa_bands);
1310      L3_imdct_gr(s->grbuf[ch], h->mdct_overlap[ch], gr_info-
          >block_type, n_long_bands);
1311      L3_change_sign(s->grbuf[ch]);
1312    }
1313  }
1314
1315  static void mp3d_DCT_II(float *grbuf, int n)
1316  {
1317    static const float g_sec[24] = {
1318        10.19000816f, 0.50060302f, 0.50241929f, 3.40760851f,
            0.50547093f, 0.52249861f, 2.05778098f, 0.51544732f,
            0.56694406f, 1.48416460f, 0.53104258f, 0.64682180f,
            1.16943991f, 0.55310392f, 0.78815460f, 0.97256821f,
            0.58293498f, 1.06067765f, 0.83934963f, 0.62250412f,
            1.72244716f, 0.74453628f, 0.67480832f, 5.10114861f};
1319    int i, k = 0;
1320  #if HAVE_SIMD
1321    if (have_simd())
1322      for (; k < n; k += 4)
1323      {
1324        f4 t[4][8], *x;
1325        float *y = grbuf + k;
1326
1327        for (x = t[0], i = 0; i < 8; i++, x++)
1328        {
1329          f4 x0 = VLD(&y[i * 18]);
1330          f4 x1 = VLD(&y[(15 - i) * 18]);
1331          f4 x2 = VLD(&y[(16 + i) * 18]);
1332          f4 x3 = VLD(&y[(31 - i) * 18]);
1333          f4 t0 = VADD(x0, x3);
1334          f4 t1 = VADD(x1, x2);
1335          f4 t2 = VMUL_S(VSUB(x1, x2), g_sec[3 * i + 0]);
```

```
1336          f4 t3 = VMUL_S(VSUB(x0, x3), g_sec[3 * i + 1]);
1337          x[0] = VADD(t0, t1);
1338          x[8] = VMUL_S(VSUB(t0, t1), g_sec[3 * i + 2]);
1339          x[16] = VADD(t3, t2);
1340          x[24] = VMUL_S(VSUB(t3, t2), g_sec[3 * i + 2]);
1341        }
1342        for (x = t[0], i = 0; i < 4; i++, x += 8)
1343        {
1344          f4 x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4], x5 = ↵
                x[5], x6 = x[6], x7 = x[7], xt;
1345          xt = VSUB(x0, x7);
1346          x0 = VADD(x0, x7);
1347          x7 = VSUB(x1, x6);
1348          x1 = VADD(x1, x6);
1349          x6 = VSUB(x2, x5);
1350          x2 = VADD(x2, x5);
1351          x5 = VSUB(x3, x4);
1352          x3 = VADD(x3, x4);
1353          x4 = VSUB(x0, x3);
1354          x0 = VADD(x0, x3);
1355          x3 = VSUB(x1, x2);
1356          x1 = VADD(x1, x2);
1357          x[0] = VADD(x0, x1);
1358          x[4] = VMUL_S(VSUB(x0, x1), 0.70710677f);
1359          x5 = VADD(x5, x6);
1360          x6 = VMUL_S(VADD(x6, x7), 0.70710677f);
1361          x7 = VADD(x7, xt);
1362          x3 = VMUL_S(VADD(x3, x4), 0.70710677f);
1363          x5 = VSUB(x5, VMUL_S(x7, 0.198912367f)); /* rotate by PI/8 */
1364          x7 = VADD(x7, VMUL_S(x5, 0.382683432f));
1365          x5 = VSUB(x5, VMUL_S(x7, 0.198912367f));
1366          x0 = VSUB(xt, x6);
1367          xt = VADD(xt, x6);
1368          x[1] = VMUL_S(VADD(xt, x7), 0.50979561f);
1369          x[2] = VMUL_S(VADD(x4, x3), 0.54119611f);
1370          x[3] = VMUL_S(VSUB(x0, x5), 0.60134488f);
1371          x[5] = VMUL_S(VADD(x0, x5), 0.89997619f);
1372          x[6] = VMUL_S(VSUB(x4, x3), 1.30656302f);
1373          x[7] = VMUL_S(VSUB(xt, x7), 2.56291556f);
1374        }
1375
1376        if (k > n - 3)
1377        {
1378  #if HAVE_SSE
1379  #define VSAVE2(i, v) _mm_storel_pi((__m64 *)(void *)&y[i * 18], v)
1380  #else /* HAVE_SSE */
1381  #define VSAVE2(i, v) vst1_f32((float32_t *)&y[i * 18], vget_low_f32    ↵
       (v))
1382  #endif /* HAVE_SSE */
1383          for (i = 0; i < 7; i++, y += 4 * 18)
1384          {
1385            f4 s = VADD(t[3][i], t[3][i + 1]);
1386            VSAVE2(0, t[0][i]);
```

```
1387            VSAVE2(1, VADD(t[2][i], s));
1388            VSAVE2(2, VADD(t[1][i], t[1][i + 1]));
1389            VSAVE2(3, VADD(t[2][1 + i], s));
1390          }
1391          VSAVE2(0, t[0][7]);
1392          VSAVE2(1, VADD(t[2][7], t[3][7]));
1393          VSAVE2(2, t[1][7]);
1394          VSAVE2(3, t[3][7]);
1395        }
1396        else
1397        {
1398  #define VSAVE4(i, v) VSTORE(&y[i * 18], v)
1399          for (i = 0; i < 7; i++, y += 4 * 18)
1400          {
1401            f4 s = VADD(t[3][i], t[3][i + 1]);
1402            VSAVE4(0, t[0][i]);
1403            VSAVE4(1, VADD(t[2][i], s));
1404            VSAVE4(2, VADD(t[1][i], t[1][i + 1]));
1405            VSAVE4(3, VADD(t[2][1 + i], s));
1406          }
1407          VSAVE4(0, t[0][7]);
1408          VSAVE4(1, VADD(t[2][7], t[3][7]));
1409          VSAVE4(2, t[1][7]);
1410          VSAVE4(3, t[3][7]);
1411        }
1412      }
1413    else
1414  #endif /* HAVE_SIMD */
1415  #ifdef MINIMP3_ONLY_SIMD
1416    {
1417    }    /* for HAVE_SIMD=1, MINIMP3_ONLY_SIMD=1 case we do not need non-intrinsic "else" branch */
1418  #else  /* MINIMP3_ONLY_SIMD */
1419    for (; k < n; k++)
1420    {
1421      float t[4][8], *x, *y = grbuf + k;
1422
1423      for (x = t[0], i = 0; i < 8; i++, x++)
1424      {
1425        float x0 = y[i * 18];
1426        float x1 = y[(15 - i) * 18];
1427        float x2 = y[(16 + i) * 18];
1428        float x3 = y[(31 - i) * 18];
1429        float t0 = x0 + x3;
1430        float t1 = x1 + x2;
1431        float t2 = (x1 - x2) * g_sec[3 * i + 0];
1432        float t3 = (x0 - x3) * g_sec[3 * i + 1];
1433        x[0] = t0 + t1;
1434        x[8] = (t0 - t1) * g_sec[3 * i + 2];
1435        x[16] = t3 + t2;
1436        x[24] = (t3 - t2) * g_sec[3 * i + 2];
1437      }
1438      for (x = t[0], i = 0; i < 4; i++, x += 8)
```

```
1439      {
1440        float x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4], x5
            = x[5], x6 = x[6], x7 = x[7], xt;
1441        xt = x0 - x7;
1442        x0 += x7;
1443        x7 = x1 - x6;
1444        x1 += x6;
1445        x6 = x2 - x5;
1446        x2 += x5;
1447        x5 = x3 - x4;
1448        x3 += x4;
1449        x4 = x0 - x3;
1450        x0 += x3;
1451        x3 = x1 - x2;
1452        x1 += x2;
1453        x[0] = x0 + x1;
1454        x[4] = (x0 - x1) * 0.70710677f;
1455        x5 = x5 + x6;
1456        x6 = (x6 + x7) * 0.70710677f;
1457        x7 = x7 + xt;
1458        x3 = (x3 + x4) * 0.70710677f;
1459        x5 -= x7 * 0.198912367f; /* rotate by PI/8 */
1460        x7 += x5 * 0.382683432f;
1461        x5 -= x7 * 0.198912367f;
1462        x0 = xt - x6;
1463        xt += x6;
1464        x[1] = (xt + x7) * 0.50979561f;
1465        x[2] = (x4 + x3) * 0.54119611f;
1466        x[3] = (x0 - x5) * 0.60134488f;
1467        x[5] = (x0 + x5) * 0.89997619f;
1468        x[6] = (x4 - x3) * 1.30656302f;
1469        x[7] = (xt - x7) * 2.56291556f;
1470      }
1471      for (i = 0; i < 7; i++, y += 4 * 18)
1472      {
1473        y[0 * 18] = t[0][i];
1474        y[1 * 18] = t[2][i] + t[3][i] + t[3][i + 1];
1475        y[2 * 18] = t[1][i] + t[1][i + 1];
1476        y[3 * 18] = t[2][i + 1] + t[3][i] + t[3][i + 1];
1477      }
1478      y[0 * 18] = t[0][7];
1479      y[1 * 18] = t[2][7] + t[3][7];
1480      y[2 * 18] = t[1][7];
1481      y[3 * 18] = t[3][7];
1482    }
1483  #endif /* MINIMP3_ONLY_SIMD */
1484  }
1485
1486  #ifndef MINIMP3_FLOAT_OUTPUT
1487  static int16_t mp3d_scale_pcm(float sample)
1488  {
1489  #if HAVE_ARMV6
1490    int32_t s32 = (int32_t)(sample + .5f);
```

```
1491    s32 -= (s32 < 0);
1492    int16_t s = (int16_t)minimp3_clip_int16_arm(s32);
1493  #else
1494    if (sample >= 32766.5)
1495      return (int16_t)32767;
1496    if (sample <= -32767.5)
1497      return (int16_t)-32768;
1498    int16_t s = (int16_t)(sample + .5f);
1499    s -= (s < 0); /* away from zero, to be compliant */
1500  #endif
1501    return s;
1502  }
1503  #else  /* MINIMP3_FLOAT_OUTPUT */
1504  static float mp3d_scale_pcm(float sample)
1505  {
1506    return sample * (1.f / 32768.f);
1507  }
1508  #endif /* MINIMP3_FLOAT_OUTPUT */
1509
1510  static void mp3d_synth_pair(mp3d_sample_t *pcm, int nch, const float
       *z)
1511  {
1512    float a;
1513    a = (z[14 * 64] - z[0]) * 29;
1514    a += (z[1 * 64] + z[13 * 64]) * 213;
1515    a += (z[12 * 64] - z[2 * 64]) * 459;
1516    a += (z[3 * 64] + z[11 * 64]) * 2037;
1517    a += (z[10 * 64] - z[4 * 64]) * 5153;
1518    a += (z[5 * 64] + z[9 * 64]) * 6574;
1519    a += (z[8 * 64] - z[6 * 64]) * 37489;
1520    a += z[7 * 64] * 75038;
1521    pcm[0] = mp3d_scale_pcm(a);
1522
1523    z += 2;
1524    a = z[14 * 64] * 104;
1525    a += z[12 * 64] * 1567;
1526    a += z[10 * 64] * 9727;
1527    a += z[8 * 64] * 64019;
1528    a += z[6 * 64] * -9975;
1529    a += z[4 * 64] * -45;
1530    a += z[2 * 64] * 146;
1531    a += z[0 * 64] * -5;
1532    pcm[16 * nch] = mp3d_scale_pcm(a);
1533  }
1534
1535  static void mp3d_synth(float *xl, mp3d_sample_t *dstl, int nch, float
       *lins)
1536  {
1537    int i;
1538    float *xr = xl + 576 * (nch - 1);
1539    mp3d_sample_t *dstr = dstl + (nch - 1);
1540
1541    static const float g_win[] = {
```

```
1542        -1, 26, -31, 208, 218, 401, -519, 2063, 2000, 4788, -5517, 7134,
               5959, 35640, -39336, 74992,
1543        -1, 24, -35, 202, 222, 347, -581, 2080, 1952, 4425, -5879, 7640,
               5288, 33791, -41176, 74856,
1544        -1, 21, -38, 196, 225, 294, -645, 2087, 1893, 4063, -6237, 8092,
               4561, 31947, -43006, 74630,
1545        -1, 19, -41, 190, 227, 244, -711, 2085, 1822, 3705, -6589, 8492,
               3776, 30112, -44821, 74313,
1546        -1, 17, -45, 183, 228, 197, -779, 2075, 1739, 3351, -6935, 8840,
               2935, 28289, -46617, 73908,
1547        -1, 16, -49, 176, 228, 153, -848, 2057, 1644, 3004, -7271, 9139,
               2037, 26482, -48390, 73415,
1548        -2, 14, -53, 169, 227, 111, -919, 2032, 1535, 2663, -7597, 9389,
               1082, 24694, -50137, 72835,
1549        -2, 13, -58, 161, 224, 72, -991, 2001, 1414, 2330, -7910, 9592,
              70, 22929, -51853, 72169,
1550        -2, 11, -63, 154, 221, 36, -1064, 1962, 1280, 2006, -8209, 9750,
               -998, 21189, -53534, 71420,
1551        -2, 10, -68, 147, 215, 2, -1137, 1919, 1131, 1692, -8491, 9863,
              -2122, 19478, -55178, 70590,
1552        -3, 9, -73, 139, 208, -29, -1210, 1870, 970, 1388, -8755, 9935,
              -3300, 17799, -56778, 69679,
1553        -3, 8, -79, 132, 200, -57, -1283, 1817, 794, 1095, -8998, 9966,
              -4533, 16155, -58333, 68692,
1554        -4, 7, -85, 125, 189, -83, -1356, 1759, 605, 814, -9219, 9959,
              -5818, 14548, -59838, 67629,
1555        -4, 7, -91, 117, 177, -106, -1428, 1698, 402, 545, -9416, 9916,
              -7154, 12980, -61289, 66494,
1556        -5, 6, -97, 111, 163, -127, -1498, 1634, 185, 288, -9585, 9838,
              -8540, 11455, -62684, 65290};
1557    float *zlin = lins + 15 * 64;
1558    const float *w = g_win;
1559
1560    zlin[4 * 15] = xl[18 * 16];
1561    zlin[4 * 15 + 1] = xr[18 * 16];
1562    zlin[4 * 15 + 2] = xl[0];
1563    zlin[4 * 15 + 3] = xr[0];
1564
1565    zlin[4 * 31] = xl[1 + 18 * 16];
1566    zlin[4 * 31 + 1] = xr[1 + 18 * 16];
1567    zlin[4 * 31 + 2] = xl[1];
1568    zlin[4 * 31 + 3] = xr[1];
1569
1570    mp3d_synth_pair(dstr, nch, lins + 4 * 15 + 1);
1571    mp3d_synth_pair(dstr + 32 * nch, nch, lins + 4 * 15 + 64 + 1);
1572    mp3d_synth_pair(dstl, nch, lins + 4 * 15);
1573    mp3d_synth_pair(dstl + 32 * nch, nch, lins + 4 * 15 + 64);
1574
1575  #if HAVE_SIMD
1576    if (have_simd())
1577      for (i = 14; i >= 0; i--)
1578      {
1579  #define VLOAD(k)                    \
```

```
1580    f4 w0 = VSET(*w++);                  \
1581    f4 w1 = VSET(*w++);                  \
1582    f4 vz = VLD(&zlin[4 * i - 64 * k]); \
1583    f4 vy = VLD(&zlin[4 * i - 64 * (15 - k)]);
1584  #define V0(k)                                 \
1585    {                                           \
1586      VLOAD(k)                                  \
1587      b = VADD(VMUL(vz, w1), VMUL(vy, w0)); \
1588      a = VSUB(VMUL(vz, w0), VMUL(vy, w1)); \
1589    }
1590  #define V1(k)                                           \
1591    {                                                     \
1592      VLOAD(k)                                            \
1593      b = VADD(b, VADD(VMUL(vz, w1), VMUL(vy, w0))); \
1594      a = VADD(a, VSUB(VMUL(vz, w0), VMUL(vy, w1))); \
1595    }
1596  #define V2(k)                                           \
1597    {                                                     \
1598      VLOAD(k)                                            \
1599      b = VADD(b, VADD(VMUL(vz, w1), VMUL(vy, w0))); \
1600      a = VADD(a, VSUB(VMUL(vy, w1), VMUL(vz, w0))); \
1601    }
1602        f4 a, b;
1603        zlin[4 * i] = xl[18 * (31 - i)];
1604        zlin[4 * i + 1] = xr[18 * (31 - i)];
1605        zlin[4 * i + 2] = xl[1 + 18 * (31 - i)];
1606        zlin[4 * i + 3] = xr[1 + 18 * (31 - i)];
1607        zlin[4 * i + 64] = xl[1 + 18 * (1 + i)];
1608        zlin[4 * i + 64 + 1] = xr[1 + 18 * (1 + i)];
1609        zlin[4 * i - 64 + 2] = xl[18 * (1 + i)];
1610        zlin[4 * i - 64 + 3] = xr[18 * (1 + i)];
1611
1612        V0(0)
1613        V2(1) V1(2) V2(3) V1(4) V2(5) V1(6) V2(7)
1614
1615        {
1616  #ifndef MINIMP3_FLOAT_OUTPUT
1617  #if HAVE_SSE
1618          static const f4 g_max = {32767.0f, 32767.0f, 32767.0f,
            32767.0f};
1619          static const f4 g_min = {-32768.0f, -32768.0f, -32768.0f,
            -32768.0f};
1620          __m128i pcm8 = _mm_packs_epi32(_mm_cvtps_epi32(_mm_max_ps
            (_mm_min_ps(a, g_max), g_min)),
1621                                         _mm_cvtps_epi32(_mm_max_ps
                    (_mm_min_ps(b, g_max), g_min)));
1622          dstr[(15 - i) * nch] = _mm_extract_epi16(pcm8, 1);
1623          dstr[(17 + i) * nch] = _mm_extract_epi16(pcm8, 5);
1624          dstl[(15 - i) * nch] = _mm_extract_epi16(pcm8, 0);
1625          dstl[(17 + i) * nch] = _mm_extract_epi16(pcm8, 4);
1626          dstr[(47 - i) * nch] = _mm_extract_epi16(pcm8, 3);
1627          dstr[(49 + i) * nch] = _mm_extract_epi16(pcm8, 7);
1628          dstl[(47 - i) * nch] = _mm_extract_epi16(pcm8, 2);
```

```
1629          dstl[(49 + i) * nch] = _mm_extract_epi16(pcm8, 6);
1630  #else  /* HAVE_SSE */
1631          int16x4_t pcma, pcmb;
1632          a = VADD(a, VSET(0.5f));
1633          b = VADD(b, VSET(0.5f));
1634          pcma = vqmovn_s32(vqaddq_s32(vcvtq_s32_f32(a),                ↵
                vreinterpretq_s32_u32(vcltq_f32(a, VSET(0)))));
1635          pcmb = vqmovn_s32(vqaddq_s32(vcvtq_s32_f32(b),                ↵
                vreinterpretq_s32_u32(vcltq_f32(b, VSET(0)))));
1636          vst1_lane_s16(dstr + (15 - i) * nch, pcma, 1);
1637          vst1_lane_s16(dstr + (17 + i) * nch, pcmb, 1);
1638          vst1_lane_s16(dstl + (15 - i) * nch, pcma, 0);
1639          vst1_lane_s16(dstl + (17 + i) * nch, pcmb, 0);
1640          vst1_lane_s16(dstr + (47 - i) * nch, pcma, 3);
1641          vst1_lane_s16(dstr + (49 + i) * nch, pcmb, 3);
1642          vst1_lane_s16(dstl + (47 - i) * nch, pcma, 2);
1643          vst1_lane_s16(dstl + (49 + i) * nch, pcmb, 2);
1644  #endif /* HAVE_SSE */
1645
1646  #else /* MINIMP3_FLOAT_OUTPUT */
1647
1648          static const f4 g_scale = {1.0f / 32768.0f, 1.0f / 32768.0f,  ↵
                1.0f / 32768.0f, 1.0f / 32768.0f};
1649          a = VMUL(a, g_scale);
1650          b = VMUL(b, g_scale);
1651  #if HAVE_SSE
1652          _mm_store_ss(dstr + (15 - i) * nch, _mm_shuffle_ps(a, a,      ↵
                _MM_SHUFFLE(1, 1, 1, 1)));
1653          _mm_store_ss(dstr + (17 + i) * nch, _mm_shuffle_ps(b, b,      ↵
                _MM_SHUFFLE(1, 1, 1, 1)));
1654          _mm_store_ss(dstl + (15 - i) * nch, _mm_shuffle_ps(a, a,      ↵
                _MM_SHUFFLE(0, 0, 0, 0)));
1655          _mm_store_ss(dstl + (17 + i) * nch, _mm_shuffle_ps(b, b,      ↵
                _MM_SHUFFLE(0, 0, 0, 0)));
1656          _mm_store_ss(dstr + (47 - i) * nch, _mm_shuffle_ps(a, a,      ↵
                _MM_SHUFFLE(3, 3, 3, 3)));
1657          _mm_store_ss(dstr + (49 + i) * nch, _mm_shuffle_ps(b, b,      ↵
                _MM_SHUFFLE(3, 3, 3, 3)));
1658          _mm_store_ss(dstl + (47 - i) * nch, _mm_shuffle_ps(a, a,      ↵
                _MM_SHUFFLE(2, 2, 2, 2)));
1659          _mm_store_ss(dstl + (49 + i) * nch, _mm_shuffle_ps(b, b,      ↵
                _MM_SHUFFLE(2, 2, 2, 2)));
1660  #else  /* HAVE_SSE */
1661          vst1q_lane_f32(dstr + (15 - i) * nch, a, 1);
1662          vst1q_lane_f32(dstr + (17 + i) * nch, b, 1);
1663          vst1q_lane_f32(dstl + (15 - i) * nch, a, 0);
1664          vst1q_lane_f32(dstl + (17 + i) * nch, b, 0);
1665          vst1q_lane_f32(dstr + (47 - i) * nch, a, 3);
1666          vst1q_lane_f32(dstr + (49 + i) * nch, b, 3);
1667          vst1q_lane_f32(dstl + (47 - i) * nch, a, 2);
1668          vst1q_lane_f32(dstl + (49 + i) * nch, b, 2);
1669  #endif /* HAVE_SSE */
1670  #endif /* MINIMP3_FLOAT_OUTPUT */
```

```
1671        }
1672      }
1673    else
1674  #endif /* HAVE_SIMD */
1675  #ifdef MINIMP3_ONLY_SIMD
1676    {
1677    }   /* for HAVE_SIMD=1, MINIMP3_ONLY_SIMD=1 case we do not need non-intrinsic "else" branch */
1678  #else /* MINIMP3_ONLY_SIMD */
1679    for (i = 14; i >= 0; i--)
1680    {
1681  #define LOAD(k)                       \
1682    float w0 = *w++;                    \
1683    float w1 = *w++;                    \
1684    float *vz = &zlin[4 * i - k * 64]; \
1685    float *vy = &zlin[4 * i - (15 - k) * 64];
1686  #define S0(k)                                                                 \
1687                                                                                \
1688      int j;                                                                    \
1689      LOAD(k);                                                                  \
1690      for (j = 0; j < 4; j++)                                                   \
1691        b[j] = vz[j] * w1 + vy[j] * w0, a[j] = vz[j] * w0 - vy[j] * w1;         \
1692    }
1693  #define S1(k)                                                                 \
1694                                                                                \
1695      int j;                                                                    \
1696      LOAD(k);                                                                  \
1697      for (j = 0; j < 4; j++)                                                   \
1698        b[j] += vz[j] * w1 + vy[j] * w0, a[j] += vz[j] * w0 - vy[j] * w1; \
1699    }
1700  #define S2(k)                                                                 \
1701                                                                                \
1702      int j;                                                                    \
1703      LOAD(k);                                                                  \
1704      for (j = 0; j < 4; j++)                                                   \
```

```
          \
1705        b[j] += vz[j] * w1 + vy[j] * w0, a[j] += vy[j] * w1 - vz[j] *
              w0; \
1706    }
1707      float a[4], b[4];
1708
1709      zlin[4 * i] = xl[18 * (31 - i)];
1710      zlin[4 * i + 1] = xr[18 * (31 - i)];
1711      zlin[4 * i + 2] = xl[1 + 18 * (31 - i)];
1712      zlin[4 * i + 3] = xr[1 + 18 * (31 - i)];
1713      zlin[4 * (i + 16)] = xl[1 + 18 * (1 + i)];
1714      zlin[4 * (i + 16) + 1] = xr[1 + 18 * (1 + i)];
1715      zlin[4 * (i - 16) + 2] = xl[18 * (1 + i)];
1716      zlin[4 * (i - 16) + 3] = xr[18 * (1 + i)];
1717
1718      S0(0)
1719      S2(1) S1(2) S2(3) S1(4) S2(5) S1(6) S2(7)
1720
1721          dstr[(15 - i) * nch] = mp3d_scale_pcm(a[1]);
1722      dstr[(17 + i) * nch] = mp3d_scale_pcm(b[1]);
1723      dstl[(15 - i) * nch] = mp3d_scale_pcm(a[0]);
1724      dstl[(17 + i) * nch] = mp3d_scale_pcm(b[0]);
1725      dstr[(47 - i) * nch] = mp3d_scale_pcm(a[3]);
1726      dstr[(49 + i) * nch] = mp3d_scale_pcm(b[3]);
1727      dstl[(47 - i) * nch] = mp3d_scale_pcm(a[2]);
1728      dstl[(49 + i) * nch] = mp3d_scale_pcm(b[2]);
1729    }
1730  #endif /* MINIMP3_ONLY_SIMD */
1731  }
1732
1733  static void mp3d_synth_granule(float *qmf_state, float *grbuf, int
        nbands, int nch, mp3d_sample_t *pcm, float *lins)
1734  {
1735    int i;
1736    for (i = 0; i < nch; i++)
1737    {
1738      mp3d_DCT_II(grbuf + 576 * i, nbands);
1739    }
1740
1741    memcpy(lins, qmf_state, sizeof(float) * 15 * 64);
1742
1743    for (i = 0; i < nbands; i += 2)
1744    {
1745      mp3d_synth(grbuf + i, pcm + 32 * nch * i, nch, lins + i * 64);
1746    }
1747  #ifndef MINIMP3_NONSTANDARD_BUT_LOGICAL
1748    if (nch == 1)
1749    {
1750      for (i = 0; i < 15 * 64; i += 2)
1751      {
1752        qmf_state[i] = lins[nbands * 64 + i];
1753      }
1754    }
```

```
1755    else
1756  #endif /* MINIMP3_NONSTANDARD_BUT_LOGICAL */
1757    {
1758      memcpy(qmf_state, lins + nbands * 64, sizeof(float) * 15 * 64);
1759    }
1760  }
1761
1762  static int mp3d_match_frame(const uint8_t *hdr, int mp3_bytes, int
        frame_bytes)
1763  {
1764    int i, nmatch;
1765    for (i = 0, nmatch = 0; nmatch < MAX_FRAME_SYNC_MATCHES; nmatch++)
1766    {
1767      i += hdr_frame_bytes(hdr + i, frame_bytes) + hdr_padding(hdr + i);
1768      if (i + HDR_SIZE > mp3_bytes)
1769        return nmatch > 0;
1770      if (!hdr_compare(hdr, hdr + i))
1771        return 0;
1772    }
1773    return 1;
1774  }
1775
1776  static int mp3d_find_frame(const uint8_t *mp3, int mp3_bytes, int
        *free_format_bytes, int *ptr_frame_bytes)
1777  {
1778    int i, k;
1779    for (i = 0; i < mp3_bytes - HDR_SIZE; i++, mp3++)
1780    {
1781      if (hdr_valid(mp3))
1782      {
1783        int frame_bytes = hdr_frame_bytes(mp3, *free_format_bytes);
1784        int frame_and_padding = frame_bytes + hdr_padding(mp3);
1785
1786        for (k = HDR_SIZE; !frame_bytes && k <
            MAX_FREE_FORMAT_FRAME_SIZE && i + 2 * k < mp3_bytes - HDR_SIZE;
              k++)
1787        {
1788          if (hdr_compare(mp3, mp3 + k))
1789          {
1790            int fb = k - hdr_padding(mp3);
1791            int nextfb = fb + hdr_padding(mp3 + k);
1792            if (i + k + nextfb + HDR_SIZE > mp3_bytes || !hdr_compare
                (mp3, mp3 + k + nextfb))
1793              continue;
1794            frame_and_padding = k;
1795            frame_bytes = fb;
1796            *free_format_bytes = fb;
1797          }
1798        }
1799        if ((frame_bytes && i + frame_and_padding <= mp3_bytes &&
1800             mp3d_match_frame(mp3, mp3_bytes - i, frame_bytes)) ||
1801            (!i && frame_and_padding == mp3_bytes))
1802        {
```

```
1803          *ptr_frame_bytes = frame_and_padding;
1804          return i;
1805        }
1806        *free_format_bytes = 0;
1807      }
1808    }
1809    *ptr_frame_bytes = 0;
1810    return mp3_bytes;
1811  }
1812
1813  void mp3dec_init(mp3dec_t *dec)
1814  {
1815    dec->header[0] = 0;
1816  }
1817
1818  int mp3dec_decode_frame(mp3dec_t *dec, const uint8_t *mp3, int
       mp3_bytes, mp3d_sample_t *pcm, mp3dec_frame_info_t *info)
1819  {
1820    int i = 0, igr, frame_size = 0, success = 1;
1821    const uint8_t *hdr;
1822    bs_t bs_frame[1];
1823    mp3dec_scratch_t scratch;
1824
1825    if (mp3_bytes > 4 && dec->header[0] == 0xff && hdr_compare(dec-
       >header, mp3))
1826    {
1827      frame_size = hdr_frame_bytes(mp3, dec->free_format_bytes) +
         hdr_padding(mp3);
1828      if (frame_size != mp3_bytes && (frame_size + HDR_SIZE > mp3_bytes
         || !hdr_compare(mp3, mp3 + frame_size)))
1829      {
1830        frame_size = 0;
1831      }
1832    }
1833    if (!frame_size)
1834    {
1835      memset(dec, 0, sizeof(mp3dec_t));
1836      i = mp3d_find_frame(mp3, mp3_bytes, &dec->free_format_bytes,
         &frame_size);
1837      if (!frame_size || i + frame_size > mp3_bytes)
1838      {
1839        info->frame_bytes = i;
1840        return 0;
1841      }
1842    }
1843
1844    hdr = mp3 + i;
1845    memcpy(dec->header, hdr, HDR_SIZE);
1846    info->frame_bytes = i + frame_size;
1847    info->frame_offset = i;
1848    info->channels = HDR_IS_MONO(hdr) ? 1 : 2;
1849    info->hz = hdr_sample_rate_hz(hdr);
1850    info->layer = 4 - HDR_GET_LAYER(hdr);
```

```
1851    info->bitrate_kbps = hdr_bitrate_kbps(hdr);
1852
1853    if (!pcm)
1854    {
1855      return hdr_frame_samples(hdr);
1856    }
1857
1858    bs_init(bs_frame, hdr + HDR_SIZE, frame_size - HDR_SIZE);
1859    if (HDR_IS_CRC(hdr))
1860    {
1861      get_bits(bs_frame, 16);
1862    }
1863
1864    if (info->layer == 3)
1865    {
1866      int main_data_begin = L3_read_side_info(bs_frame, scratch.gr_info,
           hdr);
1867      if (main_data_begin < 0 || bs_frame->pos > bs_frame->limit)
1868      {
1869        mp3dec_init(dec);
1870        return 0;
1871      }
1872      success = L3_restore_reservoir(dec, bs_frame, &scratch,
          main_data_begin);
1873      if (success)
1874      {
1875        for (igr = 0; igr < (HDR_TEST_MPEG1(hdr) ? 2 : 1); igr++, pcm +=
             576 * info->channels)
1876        {
1877          memset(scratch.grbuf[0], 0, 576 * 2 * sizeof(float));
1878          L3_decode(dec, &scratch, scratch.gr_info + igr * info-
            >channels, info->channels);
1879          mp3d_synth_granule(dec->qmf_state, scratch.grbuf[0], 18, info-
            >channels, pcm, scratch.syn[0]);
1880        }
1881      }
1882      L3_save_reservoir(dec, &scratch);
1883    }
1884    else
1885    {
1886  #ifdef MINIMP3_ONLY_MP3
1887      return 0;
1888  #else  /* MINIMP3_ONLY_MP3 */
1889      L12_scale_info sci[1];
1890      L12_read_scale_info(hdr, bs_frame, sci);
1891
1892      memset(scratch.grbuf[0], 0, 576 * 2 * sizeof(float));
1893      for (i = 0, igr = 0; igr < 3; igr++)
1894      {
1895        if (12 == (i += L12_dequantize_granule(scratch.grbuf[0] + i,
            bs_frame, sci, info->layer | 1)))
1896        {
1897          i = 0;
```

```
1898          L12_apply_scf_384(sci, sci->scf + igr, scratch.grbuf[0]);
1899          mp3d_synth_granule(dec->qmf_state, scratch.grbuf[0], 12, info-
                >channels, pcm, scratch.syn[0]);
1900          memset(scratch.grbuf[0], 0, 576 * 2 * sizeof(float));
1901          pcm += 384 * info->channels;
1902        }
1903        if (bs_frame->pos > bs_frame->limit)
1904        {
1905          mp3dec_init(dec);
1906          return 0;
1907        }
1908      }
1909  #endif /* MINIMP3_ONLY_MP3 */
1910    }
1911    return success * hdr_frame_samples(dec->header);
1912  }
1913
1914  #ifdef MINIMP3_FLOAT_OUTPUT
1915  void mp3dec_f32_to_s16(const float *in, int16_t *out, int num_samples)
1916  {
1917    int i = 0;
1918  #if HAVE_SIMD
1919    int aligned_count = num_samples & ~7;
1920    for (; i < aligned_count; i += 8)
1921    {
1922      static const f4 g_scale = {32768.0f, 32768.0f, 32768.0f,
          32768.0f};
1923      f4 a = VMUL(VLD(&in[i]), g_scale);
1924      f4 b = VMUL(VLD(&in[i + 4]), g_scale);
1925  #if HAVE_SSE
1926      static const f4 g_max = {32767.0f, 32767.0f, 32767.0f, 32767.0f};
1927      static const f4 g_min = {-32768.0f, -32768.0f, -32768.0f,
          -32768.0f};
1928      __m128i pcm8 = _mm_packs_epi32(_mm_cvtps_epi32(_mm_max_ps
          (_mm_min_ps(a, g_max), g_min)),
1929                                     _mm_cvtps_epi32(_mm_max_ps
                      (_mm_min_ps(b, g_max), g_min)));
1930      out[i] = _mm_extract_epi16(pcm8, 0);
1931      out[i + 1] = _mm_extract_epi16(pcm8, 1);
1932      out[i + 2] = _mm_extract_epi16(pcm8, 2);
1933      out[i + 3] = _mm_extract_epi16(pcm8, 3);
1934      out[i + 4] = _mm_extract_epi16(pcm8, 4);
1935      out[i + 5] = _mm_extract_epi16(pcm8, 5);
1936      out[i + 6] = _mm_extract_epi16(pcm8, 6);
1937      out[i + 7] = _mm_extract_epi16(pcm8, 7);
1938  #else  /* HAVE_SSE */
1939      int16x4_t pcma, pcmb;
1940      a = VADD(a, VSET(0.5f));
1941      b = VADD(b, VSET(0.5f));
1942      pcma = vqmovn_s32(vqaddq_s32(vcvtq_s32_f32(a),
          vreinterpretq_s32_u32(vcltq_f32(a, VSET(0)))));
1943      pcmb = vqmovn_s32(vqaddq_s32(vcvtq_s32_f32(b),
          vreinterpretq_s32_u32(vcltq_f32(b, VSET(0)))));
```

```
1944      vst1_lane_s16(out + i, pcma, 0);
1945      vst1_lane_s16(out + i + 1, pcma, 1);
1946      vst1_lane_s16(out + i + 2, pcma, 2);
1947      vst1_lane_s16(out + i + 3, pcma, 3);
1948      vst1_lane_s16(out + i + 4, pcmb, 0);
1949      vst1_lane_s16(out + i + 5, pcmb, 1);
1950      vst1_lane_s16(out + i + 6, pcmb, 2);
1951      vst1_lane_s16(out + i + 7, pcmb, 3);
1952  #endif /* HAVE_SSE */
1953    }
1954  #endif /* HAVE_SIMD */
1955    for (; i < num_samples; i++)
1956    {
1957      float sample = in[i] * 32768.0f;
1958      if (sample >= 32766.5)
1959        out[i] = (int16_t)32767;
1960      else if (sample <= -32767.5)
1961        out[i] = (int16_t)-32768;
1962      else
1963      {
1964        int16_t s = (int16_t)(sample + .5f);
1965        s -= (s < 0); /* away from zero, to be compliant */
1966        out[i] = s;
1967      }
1968    }
1969  }
1970  #endif /* MINIMP3_FLOAT_OUTPUT */
1971  #endif /* MINIMP3_IMPLEMENTATION && !_MINIMP3_IMPLEMENTATION_GUARD */
```